Wissenschaftliches Rechnen I

Hubert Grassmann, WS 2001 10. Januar 2001

Die erste Version dieses Skripts wurde im Jahre 1997 erstellt; sie ist noch **nie** *in der „neuen“ Schreibweise verfaßt worden.*

Inhaltsverzeichnis

1. http://www.informatik.uni-siegen.de/psy/docu/GoToJava2/html/cover.html
2. http://www.uni-muenster.de/ZIV/Mitarbeiter/BennoSueselbeck/Java-ws99/
3. Brandstädt, Graphen und Algorithmen, Teubner 1994
4. Flanagan, Java examples in a nutshell, O'Reilly (Köln) 1998
5. Appelrath/Ludewig, Skriptum Informatik – eine konventionelle Einführung, Teubner 1995
6. Golub/Ortega, Scientific Computing, Teubner 1996
7. Knuth, The Art of Computer Programming, Addison-Wesley 1973
8. Steyer, Java 1.2 – Schnell und sicher zum Ziel, Heyne 1998 (preiswert)
9. Krüger, Java 1.1 – Anfangen, Anwenden, Verstehen, Addison-Wesley 1997 (auch online)
10. Partl, Schlegl, Hyna, LaTeX- Kurzbeschreibung, lkurz.dvi (bei emtex)
11. Scheller, Boden, Geenen, Kampermann, Internet: Werkzeuge und Dienste, Springer 1994
12. Solymosi/Grude, Grundkurs Algorithmen und Datenstrukturen, Vieweg 2000
13. Kofler, Linux, Addison-Wesley 1998

1 Einleitung

Das Anliegen des Kurses Wissenschaftliches Rechnen I ist es zunächst, Kenntnisse zu vermitteln und Fertigkeiten zu entwickeln, die zur Nutzung von Computern zur Lösung mathematischer Probleme befähigen, und, weiter gefaßt, eine Einführung in die sinnvolle Nutzung vorhandener Computerprogramme zu geben. Dazu gehören solche Problemkreise wie

- Was kann ein Computer, was kann er nicht?
- Welche vorhandenen Programme kann man wofür nutzen?
- Was tut ein in einer bestimmten Programmiersprache vorgelegtes Programm? Wie implementiert man bekannte Algorithmen in einer Programmiersprache?
- Welche Möglichkeiten bieten Computer zur Kommunikation?
- Wie erstellt man (mathematische) Texte?

Die Antworten auf solche Fragen wollen wir in der Vorlesung und den zugehörigen Übungen sowie im Praktikum gemeinsam erarbeiten.

Vorab will ich drei Zitate anbringen, die uns zu denken geben sollten.
Das erste ist dem im Jahre 1957 erschienenen Buch „Praktische Mathematik für Ingenieure und Physiker“ von R. Zurmühl entnommen (S. 5/6):

Auch das Zahlenrechnen ist eine Kunst, die gelernt sein will. Es erfordert ständige Sorgfalt und Konzentration, und auch dann lassen sich Rechenfehler nicht ausschalten. Zu deren Aufdeckung sind, soweit irgend möglich, laufende Kontrollen in die Rechnung einzubauen, und wo es solche nicht gibt, ist doppelt zu rechnen, z.B. von zwei Personen parallel. Es empfiehlt sich das Arbeiten mit dem Bleistift, um leicht korrigieren zu können. Die Anlage gut überlegter und übersichtlicher Rechenschemata . . . hilft Fehler vermeiden und erlaubt es vor allem, die Rechnung angelernten Hilfskräften zu überlassen.

H. J. Appelrath und J. Ludewig geben in ihrem „Skriptum Informatik“ von 1995 folgende Hinweise zum Programmierstil (S. 96):

Programme sollten mit kleinstmöglichem Aufwand korrigier- und modifizierbar sein. Ein Ansatz zur Erlangung dieses Ziels ist eine hohe Lokalität durch enge Gültigkeitsbereiche. Größtmögliche Lokalität ist daher vorrangiges Ziel einer guten Programmierung! Ein Programm sollte daher folgende Merkmale aufweisen:

- *Die auftretenden Programmeinheiten (Prozeduren, Funktionen, Hauptprogramm) sind überschaubar.*
- *Die Objekte sind so lokal wie möglich definiert, jeder Bezeichner hat nur eine einzige, bestimmte Bedeutung.*
- *Die Kommunikation zwischen Programmeinheiten erfolgt vorzugsweise über eine möglichst kleine Anzahl von Parametern, nicht über globale Variablen.*

Schließlich gehen Golub und Ortega darauf ein, was ein gutes Programm ausmacht:

- *Zuverlässigkeit – Das Programm darf keine Fehler enthalten; man muß darauf vertrauen können, daß es das berechnet, was man zu berechnen beabsichtigt.*
- *Robustheit – Das Programm . . . muß in der Lage sein, . . . ungeeignete Daten zu entdecken und sie in einer den Benutzer zufriedenstellenden Art und Weise zu behandeln.*
- *Portierbarkeit – Gewöhlich erreicht man das dadurch, daß man das Programm in einer maschinenunabhängigen höheren Programmiersprache wie FORTRAN schreibt und keine „Tricks“, die sich auf die charakteristischen Eigenschaften eines speziellen Rechners stützen, benutzt . . .*
- *Wartungsfreundlichkeit – Es gibt kein Programm, das nicht von Zeit zu Zeit geändert werden muß, . . . und dies sollte mit möglichst geringer Mühe geschehen können.*

2 Zahldarstellung im Computer

Wenn im Computer ein Programm abläuft, so befinden sich im Arbeitsspeicher Informationen, die auf unterschiedliche Weise interpretiert werden: einerseits sind dies „Befehle“, die abzuarbeiten sind, andererseits „Daten“, die zu bearbeiten sind.
Was sind „Daten“?
Die kleinste Informationseinheit kann in einem Bit abgelegt werden. Ein Bit kann genau eine von zwei Informationen der Art „ja/nein“, „an/aus“, „wahr/falsch“, „0/1“ usw. enthalten.
Der kleinste adressierbare Datenbereich ist ein Byte, das sind acht Bit. Ein Byte kann also 256 verschiedene Werte annehmen.
Solche Werte können als Zahlen ($0 \leq x \leq 255$), als logischer Wert, als Druckzeichen ('A', 'B', ...), als Speicheradresse oder als Maschinenbefehl interpretiert werden.
Für viele Zwecke reicht diese Informationsmenge nicht aus, meist faßt man zwei oder vier Byte zu einem „Maschinenwort“ zusammen, ein Wort kann also $2^{16} = 65536$ bzw. $2^{32} = 4.294.967.296$ Werte annehmen.
Zur Darstellung negativer Zahlen gibt es mehrere Möglichkeiten. Wenn das höchstwertige Bit bei nichtnegativen Zahlen auf 0, bei negativen Zahlen auf 1 gesetzt wird, so schränkt dies den darstellbaren Zahlbereich auf $\pm 2^{15}$ ein. Eine anderer Möglichkeit ist es, die zur positiven Zahl `x` entgegengesetzte Zahl `x1` dadurch zu gewinnen, daß man alle Bits umkippt, dies nennt man das Einerkomplement von `x`. Zum Beispiel

```
x  = 01101100   = 64 + 32 + 8 + 4 = 108
x1 = 10010011
```

Dann gilt `x + x1 = 11111111 = 2^8 - 1`.
Wenn zum Einerkomplement noch 1 addiert (`x2 = x1 + 1 = 10010100`), so gilt
`x + x2 = 2^8 = 1|00000000 = 0`,
wenn man das neunte Bit einfach überlaufen läßt. Diese Darstellung der zu `x` entgegengesetzen Zahl hat die angenehme Eigenschaft, daß `x - y = x + y2` gilt. Die Subtraktion ist auf die Addition des sogenannten Zweierkomplements zurückgeführt worden.

Die Bearbeitung der Daten geschieht in sogenannten Registern, diese können je ein Wort (oder zwei Worte) aufnehmen, und ein Maschinenbefehl wie `ADD AX,BX` (hexadezimal: 03 C3, dezimal: 3 195) bewirkt die Addition des Inhalts des Registers `BX` zum Inhalt von `AX`. Neben arithmetischen Operationen (Addition, Subtraktion, Multiplikation, Division mit Rest) stehen z.B. bitweise Konjunktion, Shift-Operation usw. zur Verfügung.
Um das Erstellen eigener Programme zu erleichtern, wurden seit den 60er Jahren „höhere“ Programmiersprachen entwickelt. Sie sollten den Programmierer davon entlasten, den Maschinencode kennen zu müssen, die Adressen der Daten zu verwalten, komplexe Datenstrukturen (mehr als nur Zahlen) zu verwalten sowie häufig angewandte Befehlsfolgen bequem aufzurufen.
Wir geben hier eine Ahnentafel der gebräuchlichsten Programmiersprachen an (nach Appelrath/Ludewig);

1945	Plankalkül (K. Zuse für Z3)				
1952	Assembler		1970	Prolog	Pascal
1956	Fortran [1]		1974	C	
1959	Algol60		1975	APL2	
1960	LISP	COBOL [2]	1977	Fortran77	MODULA-2
1962	APL [3]	BASIC	1980	Smalltalk-80	ADA
1964	PL1		1987	C++	
1967	SIMULA		1988		Oberon
1968	Algol68		1990	Fortran90	
1969	Logo		1995	Java	

Als nutzbarer Zahlbereich stehen also nur die Zahlen zwischen 0 und 65355 oder, wenn man das höchste Bit als Vorzeichenbit interpretiert, zwischen -32768 und 32767 zur Verfügung, das sind die Zahlen, deren Datentyp man als `INTEGER` bezeichnet. Da dies, außer bei der Textverarbeitung und ähnlichen Problemen, nicht ausreicht, stellen höhere Programmiersprachen zusätzlich Gleitkommazahlen zur Verfügung, die wie bei Taschenrechnern in der Form 1.23456E-20 dargestellt werden, dabei ist die Mantisse meist auf sechs bis acht Dezimalstellen und der Exponent auf Werte zwischen -40 und 40 beschränkt. Diese Schranken sind stark Hardware- und implementationsabhängig, auch 16-stellige Genauigkeit und Exponenten bis ± 317 sind möglich. Es gibt hier Standards, an die sich eigentlich jeder Computerhersteller halten müßte, darauf gehen wir nun ein.

Beim Rechnen mit Computerzahlen treten einige Probleme auf, deren elementarste wir hier besprechen wollen:

```
x:= 40000, y:= 30000, z:= 20000
x + y:      -30536
x + y - z:   25000
int(x+y-z): -15536
```

Die fehlerhafte Berechnung von x + y ist durch den Überlauf ($x + y \geq 2^{15}$) begründet, wenn sich das Resultat im „richtigen“ Bereich befindet, wird alles wieder gut. Solche Fehler können während des Programmlaufs erkannt werden, wenn bei der Übersetzung eine Bereichsüberprüfung eingestellt wird. Nach dem Standard darf so etwas nicht passieren.

Auf alle Fälle wird 20000 - 10000 + 15000 richtig berechnet, das Kommutativgesetz der Addition ist also für Computerzahlen nicht gültig, das Gleiche gilt für das Assoziativgesetz.
Wir kommen nun zu den Gleitkommazahlen (der Typ-Name „real“ ist irreführend). Hierzu ist zunächst folgendes zu sagen:

1. Es gibt nur endlich viele Gleitkommazahlen.

[1] formula translator
[2] common business oriented language
[3] a programming language

2. Die Bilder der vorhandenen Gleitkommazahlen sind auf der Zahlengeraden unterschiedlich dicht. Wenn z.B. mit 7stelliger Mantisse gerechnet wird, so ist die Nachbarzahl von $1.0 \cdot 10^{-10}$ die Zahl $1.000001 \cdot 10^{-10}$, der Unterschied ist 10^{-16}. Die Nachbarzahl von $1.0 \cdot 10^{10}$ ist $1.000001 \cdot 10^{10}$, der Unterschied ist 10^5. Das hat folgende Konsequenzen:

3. Das Addieren einer kleinen Zahl zu einer großen ändert diese evtl. nicht (das macht aber nichts).

4. Die Subtraktion fast gleichgroßer Zahlen kann zu erheblichen Fehlern führen; da sich die führenden Ziffern gegenseitig auslöschen, werden hinten Nullen nachgeschoben. Dieser in der Dualdarstellung der Zahlen noch sichtbare Effekt wird aber nach der Konvertierung in die Dezimaldarstellung verschleiert. Deshalb lautet eine goldene Regel:

   Vermeidbare Subtraktionen fast gleichgroßer Zahlen sollen vermieden werden.

Wir wollen einige Effekte vorstellen, die beim Rechnen mit Gleitkommazahlen auftreten: Die Terme

$$99 - 70\sqrt{2} = \frac{1}{99 + 70\sqrt{2}} = \frac{1}{(1+\sqrt{2})^6}$$

ergeben bei (dezimaler) 10-stelliger Genauigkeit die Werte
0.0050506600000, 0.005050633885, 0.005050633890,
bei 5-stelliger Genauigkeit erhalten wir
0.006, 0.0050507, 0.0050508,
bei 3-stelliger Genauigkeit (so genau konnte man mit einem Rechenschieber rechnen)
0.300, 0.00506, 0.00506.
Mein Taschenrechner mit 8stelliger Genauigkeit liefert in allen drei Fällen dasselbe: 0.0050506.

Um die Subtraktion fast gleichgroßer Zahlen zu vermeiden, gibt es ein paar Tricks:

$$\frac{1}{x} - \frac{1}{x+1} = \frac{1}{x(x+1)}$$

$$\frac{1}{x+1} - \frac{1}{x-1} = \frac{2}{x(x^2-1)} + \frac{2}{x}$$

$$\sqrt{x+1} - \sqrt{x} = \frac{1}{\sqrt{x+1} + \sqrt{x}}$$

Wir berechnen noch die Summe $\sum_{k=1}^{n} \frac{1}{k}$, und zwar einmal vorwärts und einmal rückwärts:

n	vorwärts	rückwärts
10	2.928 968 254	... 54
100	5.187 377 520	... 19
1000	7.485 470 857	... 65

Das zweite Ergebnis ist genauer, da zuerst die kleinen Zahlen addiert werden. Zu bemerken ist außerdem, daß diese divergente Reihe auf Computern konvergiert (Das Ergebnis liegt etwa bei 14).

2.1 Gleitkommazahlen und der IEEE-Standard

Gleitkommazahlen werden in einer Form dargestellt, die der Exponentenschreibweise $\pm m \cdot 10^e$ ähnelt. Anstelle der Basis 10 für Dezimalzahlen wird die Basis 2 verwendet, wenn ein Maschinenword von 32 Bit zur Verfügung steht, so könnte ein Bit für das Vorzeichen (0 für +, 1 für −), acht Bit für den Exponenten verwendet werden. Wenn negative Exponenten in Zweierkomplementdarstellung geschrieben werden, sind als Werte hierfür – 128 bis 127 möglich. Die restlichen 23 Bit stehen für die Mantisse zur Verfügung:

$$x = b_0.b_1 \dots b_{22}$$

Die Darstellung heißt normalisiert, wenn $b_0 = 1$, also $1 \leq x < 2$ ist.
Wir überlegen, warum die normalisierte Darstellung Vorteile hat:

$$\frac{1}{10} = (0.0001100110011\dots)_2$$

Da wir den unendlichen 2er-Bruch abschneiden müssen, behalten wir möglichst viele signifikante Stellen, wenn wir auf die führenden Nullen verzichten. Die normalisierte Darstellung von $\frac{1}{10}$ ist $1.100110011\dots \cdot 2^{-4}$. Für die Zahl Null $= 0.0\dots0$ exisiert keine normalisierte Darstellung.

Die Lücke zwischen der Zahl 1 und der nächstgrößeren Zahl heißt die Maschinengenauigkeit ϵ; dies ist die kleinste Mashinenzahl, die bei der Addition zur 1 etwas von 1 verschiedenes ergibt. Hier ist also $\epsilon = 2^{-22}$. Wenn wir mit $\times \in \{+,\ -,\ \cdot,\ :\}$ eine Rechenoperation mit exakten Zahlen und mit $\otimes$ die entsprechende Rechenoperation mit Maschinenzahlen bezeichnen, so gilt

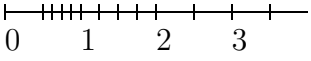
$$a \otimes b = (a \times b) \cdot (1 + k) \text{ mit } |k| < \epsilon.$$

Wir veranschaulichen die Verhältnisse, indem wir uns einen Spielzeugcomputer vorstellen, wo für jede Mantisse 3 Bit zur Verfügung stehen und der Exponent die Werte $-1, 0, 1$ annehmen kann, also

$$b_0.b_1b_2 \cdot 2^e.$$

Die größte darstellbare Zahl ist $(1.11)_2 \cdot 2^1 = 3.5$, die kleinste positive ist $(1.00)_2 \cdot 2^{-1} = \frac{1}{2}$.
Insgestamt sind die folgenden und die zu ihnen entgegengesetzten darstellbar:

$0, \frac{1}{2}, \frac{5}{8}, \frac{6}{8}, \frac{7}{8}, 1, 1\frac{1}{4}, 1\frac{1}{2}, 1\frac{3}{4}, 2, 2\frac{1}{2}, 3, 3\frac{1}{2}.$



Die Maschinengenauigkeit ist $\epsilon = 0.25$. Die Lücke zwischen der Null und der kleinsten positiven Zahl ist viel größer als die Lücken zwischen den kleinen positiven Zahlen.

Die oben genannte Darstellung von Zahlen wurde bei Rechnern der IBM 360/370 - Serie in den 70ger Jahren verwendet. Auf der VAX wurde die Maschinengenauigkeit dadurch halbiert (2^{-23}), daß man die führende 1 in der normalisierten Darstellung wegließ und somit ein Bit für die Mantisse gewann.
Bei der Darstellung der Null ging das allerdings nicht, denn hier gibt es keine führende 1 und .000000 bedeutet eben die Zahl 1. Hier mußte also eine Sonderregelung getroffen

werden. Darüberhinaus ist es sinnvoll, die Lücken um die Null zu füllen. Für diese neue Darstellung wird das Exponentenfeld mitgenutzt, was die maximal möglichen Exponenten etwas einschränkt.
Die im folgenden behandelte Darstellung wurde 1985 als IEEE-Standard (Institute for Electrical and Electronis Engineers) entwickelt. Dieser Standard wird von den führenden Chipherstellern Intel und Motorola sorgfältig eingehalten.

Die Bedeutung der Bitfolgen $\pm a_1 \dots a_8\, b_1 \dots b_{23}$ ist aus folgender Tabelle abzulesen:

$a_1 \dots a_8$	numerischer Wert
$0 \dots 00 = 0$	$0.b_1 \dots b_{23} \cdot 2^{-126}$
$0 \dots 01 = 1$	$1.b_1 \dots b_{23} \cdot 2^{-126}$
$0 \dots 10 = 2$	$1.b_1 \dots b_{23} \cdot 2^{-125}$
$\dots$	$\dots$
$01 \dots 0 = 127$	$1.b_1 \dots b_{23} \cdot 2^{0}$
$10 \dots 0 = 128$	$1.b_1 \dots b_{23} \cdot 2^{1}$
$\dots$	$\dots$
$11 \dots 10 = 254$	$1.b_1 \dots b_{23} \cdot 2^{127}$
$11 \dots 11 = 255$	$\pm\infty$, wenn $b_1 = \dots = b_{23} = 0$
	„not a number“ sonst

Bemerkungen

1. Der Zahlbereich umfaßt etwa 10^{-38} bis 10^{38}, die Genauigkeit beträgt 6 bis 7 Dezimalstellen.

2. Anstelle des Exponenten e wird $e + 127$ gespeichert (biased representation, beeinflußte Darstellung).

3. Die Zahlen in der ersten Zeile füllen die Lücke zwischen 0 und den kleinsten normalisierten Zahlen, diese Zahlen heißen subnormal. Die Genauigkeit subnormaler Zahlen ist geringer als die normalisierter.

4. Der Vergleich zweier Zahlen kann durch bitweisen Vergleich von links nach rechts durchgeführt werden, wobei man beim ersten Unterschied abbrechen kann.

5. Durch Einführung des Symbols ∞ kann bei Division durch Null ein definierter Zustand hergestellt werden.

6. Die angegebene Darstellung heißt IEEE single precision, daneben gibt es double precision (64 Bit) und extended precision (auf PCs 80 Bit).

2.2 Runden

Sei x eine reelle Zahl (ein unendlicher Zweierbruch); mit x_- wird die nächstkleinere Gleitkommazahl bezeichnet, sie entsteht durch Abschneiden nach dem 23. Bit. Mit x_+ wird die nächstgrößere Gleikommazahl bezeichnet. Wenn $b_{23} = 0$ ist, so setzt man $b_{23} = 1$. Wenn aber $b_{23} = 1$ ist, so erhält man x_+ durch einige Bitüberträge.

Die zur reellen Zahl x gerundete Gleitkommazahl ist entweder x_- oder x_+, jenachdem, ob nach oben oder unten oder zur nächstliegenden Zahl gerundet werden soll (letzteres ist am meisten verbreitet). Falls die Entscheidung unentschieden ausgeht, ist die Zahl zu wählen, deren letztes Bit gleich 0 ist.

Bei Multiplikationen/Divisionen mit Gleitkommazahlen ist das exakte Ergebnis oft keine Gleitkommazahl; der IEEE-Standard fordert, das das Resultat der korrekt gerundete Wert des exakten Ergbenisses ist. Dazu ist es z.B. bei Subtraktionen nötig, mit mindestens einem zusätzlichen Bit (nach dem 23.) zu rechnen; wir berechnen als Beispiel $1.00 \cdot 2^0 - 1.11 \cdot 2^{-1} = 1.00 \cdot 2^{-3}$ mit dreistelliger Mantisse:

```
  1.00          1.00
- 0.11|1      - 0.11
--------     -------
  0.00|1        0.01
```

Wenn die 1 nach dem senkrechten Strich weggelassen werden wird, erhält man das falsche Ergebnis 2^{-2}.
Dieses zusätzliche Bit war ursprünglich bei IBM-360-Rechnern nicht vorgesehen und ist es heute auf Cray-Rechnern immer noch nicht. Die Register einer korrekt arbeitenden Maschine, etwa im 387er Koprozessor, im 486 DX oder im Pentium sind also breiter als 32 Bit. Wenn kein Koprozessor vorhanden ist, so wird die Gleitkommaarithmetik vom verarbeitenden Programm emuliert und man ist zunächst nicht sicher, ob dabei der IEEE-Standard eingehalten wird. Weiter ist zu beachten, daß die in Registern vorhandenen Zahlen eine höhere Genauigkeit besitzen, als in einer Gleitkommavariablen abgespeichert werden kann. Solche Effekte werden wir in den Übungen behandeln: Vermeiden Sie Tests wie

```
if (a == 0)
```

oder

```
do while (a > 0)
```

sondern führen Sie Funktionen `boolean null(a)` oder `boolean pos(a)` ein; dabei gelangen die Registerinhalte in den Speicher und eine „unausgeglichene“ Arithmetik wird verhindert.

2.3 Ausnahmebehandlung: Divsion durch Null

In den 50er Jahren gab man als Ergebnis einer Division durch Null einfach die größte darstellbare Zahl aus und hoffte, daß der Nutzer dann schon merken würde, daß da etwas nicht stimmt. Das ging aber so oft schief ($\infty - \infty = 0$), daß man später dazu überging, das Programm mit einer Fehlermeldung abzubrechen. Das muß aber nicht sein:
Der Gesamtwiderstand R zweier parallelgeschalteter Widerstände R_1, R_2 ist gleich

$$R = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2}}.$$

Wenn $R_1 = 0$ ist, so ist

$$R = \frac{1}{\frac{1}{0} + \frac{1}{R_2}} = \frac{1}{\infty + \frac{1}{R_2}} = \frac{1}{\infty} = 0,$$

was ja auch sinnvoll ist.
Im IEEE-Standard sind sinnvolle Ergebnisse für $\frac{1}{0} = \infty, x + \infty = \infty, \ldots$, aber $\infty \cdot 0, 0/0, \infty - \infty, \infty/\infty$ liefern Ergebnisse vom Typ NaN (not a number), und weitere Operationen mit NaNs ergeben NaNs, so daß man den Fehler erfolgreich suchen kann.

Zu Abschluß schauen wir uns an, wie ein Pascal-Programm die Werte $\dfrac{1}{\sqrt{a+b} - \sqrt{a}}$ und $\dfrac{\sqrt{a+b} + \sqrt{a}}{b}$, die übereinstimmen, für $a = 1000,\ b = 0,001$ berechnet, und zwar mit den Zahltypen `real, single und extended` (bei der Verwendung von `double` erfolgte ein Laufzeitfehler: Division durch Null).

```
real        63245.56901478767400
            63245.56901478767400
single      63245.56640625000000
            63245.56640625000000
extended    63245.56901475061060
            63245.56901475193460
```

Bei Fortran unterscheiden sich bei `real*4`-Arithmetik die Ergebnisse bereits in der zweiten Dezimalstelle, bei `real*8`-Arithmetik in der letzten angezeigten:
`63245.56601074943`
Bei Java gibt es folgende Ergebnisse:
63245.56901639513
63245.569014751934
1.6431949916295707E-6
den ganzzahligen Teil subtrahiert:
0.569016395129438
0.5690147519344464
1.6431949916295707E-6
aber es kommen Zweifel auf, ob die dritte Nachkommastelle überhaupt zuverlässig ist. Mit einem Taschenrechner habe ich 63243.107 (!) bzw. 63245.569 „berechnet“. Man fragt sich, welches denn nun eigentlich das „richtige“ Ergebnis ist. Hier hat aber Java die Nase vorn, da es auch eine „lange“ Arithmetik bereitstellt.

Paarweise Addition

Wenn viele Zahlen $a_1, \ldots, a_n$ zu addieren sind und man das naheliegende Verfahren $s = s + a_i,\ i = 1, \ldots, n$ verwendent, so wird, wenn etwa a_1 die restlichen Summanden deutlich dominiert, das Ergebnis fast gleich a_1 sein, selbst wenn die Summe der restlichen Terme — für sich genommen — einen bedeutenden Beitrag liefer würde. In diesem Fall wäre es besser, $a_1 + (a_2 + \ldots + a_n)$ zu berechnen, da das Assoziativgesetzt nicht gilt. Die beschriebene Situation kann man natürlich nicht voraussehen.

Das Ergebnis könnte genauer sein, wenn wir schrittweise benachbarte Summanden addieren:

$$((a_1 + a_2) + (a_3 + a_4)) + \cdots$$

Noch besser könnte es gehen, wenn die Paare jeweils betragsmäßig dieselbe Größenordnung hätten, die Folge der a_i also zuvor sortiert würde. Eine genauere Analyse dieser Situation wird im 2. Semester vorgenommen.
Wie man soetwas organisieren kann, werden wir im Lauf des Semesters sehen. Es wird dann ein höherer Organisationsaufwand betrieben (was Zeit kostet), aber die Genauigkeit des Ergebnisses kann eigentlich nicht schlechter werden.
Ähnliche Effekte treten beim symbolischen Rechnen (bei Computeralgebrasystemen) auf; hier treten keine Genauigkeits-, sondern Zeit- und Speicherplatzprobleme auf. Auch hier muß man bei komplexen Algorithmen nachschauen, welche Operation zu welchen Zeitpunkt ausgeführt bzw. unterlassen werden sollte. Auch dazu später mehr.

Linux

Wir werden die Übungen und das Praktikum am Rechner durchführen, und zwar unter dem Betriebssystem Linux (von Linus Torvalds, Helsinki 1991). Hier soll kurz eine Übersicht über die wichtigsten Kommandos gegeben werden:

`passwd`	ändern des Paßworts
`man kommando`	zeigt das Manual zum kommando
`cd name`	wechselt in das Unterverzeichnis `name`
`cd ..`	wechselt in das übergeordnete Verzeichnis
`cd`	wechselt in das Heimatverzeichnis
`more name`	zeigt den Inhalt der Datei `name`
`ls`	zeigt Inhaltsverzeichnis
`ls -al`	zeigt Inhaltsverzeichnis detailliert
`ls -al | more`	zeigt Inhaltsverzeichnis detailliert, jeweils ein Bildschirm
`mkdir a`	erstellt Unterverzeichnis `a`
`cp a b`	kopiert die Datei `a` in die Datei `b`
`mv a b`	verschiebt die Datei `a` in die Datei `b`
`rm a`	löscht `a` (unwiederbringlich)
`chmod`	ändert Zugriffsrechte

Das Kommando `chmod` bestimmt die Zugriffsrechte für den User (u), die Gruppe (g), und alle Welt (a); die Rechte sind Lesen (r = 4), Schreiben (w = 2) und ausführen (x = 1) (bei Verzeichnissen bedeutet x: Zugang).
Also `chmod 744 *` bedeutet: ich kann alles, alle können nur lesen.

Die Tastatur kann man sich (unter dem X-System) umdefinieren, indem man in die Datei `.xinitrc` zum Beispiel die Zeile
`xmodmap -e '' keycode 79 = braceleft ''`
einträgt. Dadurch erhält man die geschweifte öffnende Klammer auf der Taste 7 im numerischen Block (allerdings erst beim nächsten Einloggen). Die Codes der Tasten:

7 hat 79; 8 hat 80; 9 hat 81;
4 hat 83; 5 hat 84; 6 hat 85;
1 hat 87; 2 hat 88; 3 hat 89.

3 Java-Grundlagen

Vorbemerkungen:
Java ist nicht „einfach“ zu erlernen, es ist eine „richtige“ Programmiersprache, deren Regeln man sich erarbeiten muß.
Java-Programme werden in einen „Bytecode“ übersetzt, zur Ausführung ist ein Interpreter notwendig. Daraus resultiert das Vorurteil, daß Java langsam sei. Daran ist etwas Wahres: Rechenintensive Programme können im Vergleich zu C-Programmen bis zum 10-fachen der Rechenzeit benötigen. Daran arbeiten die Java-Entwickler. „Viele Beobachter gehen davon aus, daß Java-Programme in 2-3 Jahren genausoschnell wie C/C++-Programme laufen.“ Dieses Zitat stammt aus dem Jahr 1997.
Das Abtract Windowing Toolkit (AWT), das Graphik-Routinen bereitstellt, werden wir manchmal nutzen, ohne daß hier auf dessen Funktionalität eingegangen wird.

Die zu bearbeitenden Daten werden als „Variablen“ bezeichnet (dies hat nichts mit dem gleichnamigen Begriff aus der Analysis zu tun!), „Konstanten“ sind spezielle Variable, deren Wert beim Programmstart feststeht und der nicht mehr geändert werden kann.
Jede im Programm verwendete Variable hat einen Namen, der aus den Zeichen A, B, ... , Z, a, b, ... z, 0, ... , 9 und dem Unterstrich zusammengesetzt ist; dabei darf das erste Zeichen keine Ziffer sein.
Vor einer ersten Verwendung einer Variablen muß der „Typ“ der Variablen festgelegt werden (die Variable wird „deklariert“). Der Typ einer Variablen legt einerseits fest, wieviel Speicherplatz hierfür bereitgestellt werden soll; andererseits wird anhand der Typen während der Übersetzung festgestellt, ob mit den Operanden auszuführende Operationen erlaubt sind, eine Typunverträglichkeit führt zu einem Syntaxfehler. Dies ist für den Programmierer oft ärgerlich, aber immer hilfreich, weil sich bei Syntaxfehlern (neben Schreibfehlern) oft echte Programmierfehler entdecken lassen. Allerdings sind verschiedene Programmiersprachen unterschiedlich streng bei der Typ-Prüfung. Manche Sprachen lassen sogar arithmetische Operationen zwischen Zahlen und Wahrheitswerten zu. Auf der anderen Seite ist manchmal eine Typ-Umwandlung notwendig, hierzu werden spezielle Funktionen bereitgestellt (`(float) 1.0`), oder die Programmierer greifen zu „dirty tricks“, wobei sie bei Java auf Granit beißen. Viele Fehler, die z.B. beim Programmieren in C++ dadurch entstehen, daß der Programmierer für die Speicherverwaltung selbst zuständig ist, können bei Java nicht mehr auftreten. Es gibt keine Pointer.

3.1 Primitive Typen

Bei Java stehen die folgenden Grundtypen zur Verfügung (alle Zahlen sind vorzeichenbehaftet):

Typ	Größe	Bemerkung
byte	1 Byte	-128 ... 127
short	2 Byte	- 32768 ... 32767
int	4 Byte	2 Milliarden
long	8 Byte	92..07 $\sim 10^{19}$ (ÜA)
		bisher alle ganzzahlig
float	4 Byte	Gleitkommazahl, IEEE 754-1985 Standard
double	8 Byte	max. 10^{317}
char	2 Byte	Unicode-Zeichensatz, noch lange nicht ausgefüllt
boolean	1 Bit	`true` oder `false`

Wir werden den Typ `float` nicht verwenden.

3.2 Operatoren

Mit Hilfe von Operatoren kann man aus Variablen Ausdrücke zusammensetzen, es stehen u.a. die arithmetischen Operatoren

`+, -, *, /, %`

zur Verfügung (`%` ist der modulo-Operator), darüberhinaus gibt es Zuweisungsoperatoren (man spart Schreibarbeit, aber es wird etwas unübersichtlich), den Inkrementoperator `++`, den Dekrementoperator `--`, Bitoperatoren, Verschiebungsoperatoren, die Vergleichsoperatoren

`==, !=, <, >, <=, >=`

sowie die logischen Operatoren `&&, ||, !`. Wenn in einem Ausdruck mehrere logische Operatoren vorkommen, so werden die Ergebnisse von links nach rechts ausgewertet. Die Auswertung wird abgebrochen, sobald das Ergebnis feststeht (das Ergebnis einer `||` -Operation ist `true` oder das Ergebnis einer `&&`-Operation ist `false`); das ist nützlich, wenn z.B. weiter rechts stehende Operanden gar nicht existieren.
Bei Java werden die Operatoren in Ausdrücken, in denen unterschiedliche Zahltypen vorkommen, automatisch in den den übergeordneten Zahltyp konvertiert. Das sollte man ausprobieren. Es darf nämlich nicht falsch verstanden werden! Ein häufiger Fehler ist

```
double x;
x = 1 / 2;
```

Nun, das ist völlig korrekt, es wird nur nicht das berechnet, was man vielleicht denkt: x ist eine Gleitkommazahl, also wird das Ergebnis gleich 0.5 sein. Nein, 1 und 2 sind ganzzahlig (im Gegensatz zu 1.0 und 2.0) und der Term 1/2 wird ganzzahlig berechnet, ist also gleich 0. An diese Stelle gehört ein ganz großes Ausrufungszeichen.

Ein Programm setzt sich aus Anweisungen zusammen, z.B.

`a = b + c; x = y * z;`

Rechts vom Gleichheitszeichen steht ein Ausdruck, links eine Variable, die den Wert des berechneten Ausdrucks erhält.
Einer Charakter-Variablen weist man mit `a = 'a'` einen konstanten Wert zu; bei einer String-Variablen (einer Zeichenkette) sind die Anführungszeichen zu verwenden:

`s = "abc"`; numerische Konstanten werden als Gleitkommazahlen gedeutet, wenn irgendetwas darauf hinweist, das es welche sein sollen, z.B. `3.14, 2f, 0F, 1e1, .5f, 6.`; wenn `f` oder `F` (für `float`) fehlt, aber ein Dezimalpunkt oder ein Exponent auf eine Gleitkommazahl hinweist, wird `double` angenommen.
Obwohl nicht alle Operatoren eingeführt wurden, soll hier die Vorrangs-Reihenfolge dargestellt werden; wenn davon abgewichen werden soll, sind Klammern zu setzen:
`. [] ()` *einstellig:* `+ - ! ++ -- instanceof`
zweistellig: `* / % + - < <= >= > == != & ^ | && || ?: =`
Variablen, die einem der bisher genannten primitiven Datentypen zugehören, stehen nach ihrer Deklaration sofort zur Verfügung, können also belegt und weiter verarbeitet werden. Die Deklarationen müssen nicht, wie in anderen Sprachen, am Anfang eines Blocks stehen, jedenfalls aber vor der ersten Verwendung.
Bei einer Division durch Null entsteht der Wert $\pm\infty$, wenn der Zahlbereich überschritten wird, ensteht `NaN`.

3.3 Felder

Ein Feld (array) ist eine Folge von Variablen ein- und desselben Datentyps (der nicht primitiv sein muß, ein Feld von gleichartigen Feldern ist eine Matrix usw.). Um mit einem Feld arbeiten zu können, muß es

1. deklariert werden,

   ```
   int folge[];
   int[] folge;
   double[][] matrix;
   ```

2. Speicherplatz erhalten,

   ```
   folge = new int[5];
   ```

   damit stehen `folge[0], ... , folge[4]` zur Verfügung,

3. gefüllt werden:

   ```
   folge[1] = 7;
   ```

Eine andere Möglichkeit besteht darin, das Feld sofort zu initialisieren:
`int folge[] = {1,2,3,4,5};`
dann gibt `folge.length` die Länge an.

3.4 Programmsteuerung

Um den Programmfluß durch die bearbeiteten Daten steuern zu können, benötigt man Auswahlanweisungen; es gibt die `if`-, `if-else`- und die `switch`-Anweisung.

```
if ((a != b) || (c == d))
{
  a:= 2; // hier sind 2 Anweisungen zu einen Block zusammengefasst worden
```

```
  b:= c + d;
}
else
  a:= 0;
```

Die Auswertung der logischen Bedingungen erfolgt in der angegebenen Reihenfolge; es wird höchstens einer der Blöcke abgearbeitet. Der `else`-Zweig kann entfallen.

Zählergesteuerte Wiederholungsanweisung:
Syntax:

```
  for (init; test; update) anweisung;
```

```
for (i = 1; i <= 10; i++)     for (i = a.length - 1, i >= 0; i--)
{                               s = s + a[i];
  a = a * a;
  b = b * a;
}
```

Kopfgesteuerte Wiederholungsanweisung:
Syntax:

```
  while (condition) anweisung;
```

```
while (true)
  a = a * a;        // dies ist eine Endlosschleife
```

Die while-Schleife wird solange wiederholt, wie die angegebene Bedingung wahr ist. Innerhalb der Schleife sollte etwas getan werden, was den Wahrheitswert der Bedingung ändert.

Fußgesteuerte Wiederholungsanweisung:
Syntax:

```
 do anweisung; while (condition);
```

```
do
{
  a = a * a;
  b = b * a;
}
while (b < 20);
```

Die `do`-Schleife ist nicht-abweisend, sie wird mindestens einmal durchlaufen.
Schleifen können auch geschachtelt werden:

```
  for (i = 1; i <= 10; i++)
    for (j = 1; j <=10; j++)
      a[i][j] = b[j][i];
```

Ich habe mit Pascal den Test gemacht, ob (bei nicht-quadratischen Feldern) die Reihenfolge des Durchlaufs der Schleifenvariablen eine Auswirkung auf die Rechenzeit hat und keinen Unterschied feststellen können. Man sollte aber beachten: In `for`-Schleifen wird normalerweise auf indizierte Variable zugegriffen und die Berechnung der Adresse eines Feldelements aus dem Index braucht Zeit. Man sollte solche Dereferenzierungen so selten wie möglich durchführen.

Es ist sicher ein schlechter Progammierstil, wenn man gezwungen ist, die Berechnung innerhalb einer Schleife abzubrechen oder die Schleife ganz zu verlassen. Für diesen Notfall stehen die Anweisungen `continue` und `break` zur Verfügung, die auch noch mit Sprungmarken versehen werden können (das erinnert doch sehr an die `goto`-Möglichkeiten, mit der man sich Programm-Labyrithe schaffen kann).
Schließlich sind noch Anweisungen zur Ein- und Ausgabe von Daten notwendig.

```
System.out.print(Zeichenkette);
```

gibt die `Zeichenkette` aus. Zeichenketten lassen sich mit dem +-Operator verknüpfen: wenn `a` eine numerische Variable ist, so wird sie innerhalb der `print`-Anweisung mittels `" " + a` in eine Zeichenkette umgewandelt und ausgegeben. Wenn `println` verwendet wird, so erfolgt ein Zeilenvorschub.
Die Eingabe ist etwas schwieriger: Hier müssen mögliche Fehler explizit abgefangen werden (es muß aber nichts getan werden): Wir lesen eine ganze Zahl, indem wir eine Zeichenkette lesen und diese umwandeln:

```
import java.io.*;
public static int readint()
{
  String s; int i, k;
  DataInput d = new DataInputStream(System.in);
  try
  {
    s = d.readLine();
    i = 0;
    for (k=0; k < s.length(); k++)
      i = 10 * i + ((int)s.charAt(k) - 48);
    return i;
  }
  catch (IOException ignored) {}
  return(0);
}
```

Die Berechnung von `i` in der `for`-Schleife kann durch
`i = Integer.parseInt(s)`
ersetzt werden.

```
import java.io.*;
public class cat
{
  public static void main(String args[])
```

```
  {
    DataInput d = new DataInputStream(System.in);
    String line;
    try
    {
      while ((line = d.readLine()) != null)
      System.out.println(line);
    }
    catch (IOException ignored){}
  }
}
```

Noch ein Beispiel für das Rechnen mit beliebig langen Zahlen:

```
import java.math.*;
import java.util.*;
import java.io.*;
class v2
{
  public static BigInteger lies() // soll eine lange Zahl lesen
  {
    BigInteger a;
    String s;
    s = B.readstr();
    a = new BigInteger(s);
    return(a);
  }
  public static void schreib(BigInteger a)
  {
    String s = a.toString();
    System.out.println(" "+s);
  }
public static void main(String args[])
{
  BigInteger a,b,c;
  a = lies();
  schreib(a);
  b = lies();
  schreib(b);
  c = a.multiply(b);
  schreib(c);
}
}
```

3.5 Methoden

Rechenabläufe, die mehrmals (mit verschiedenen Parametern) wiederholt werden, oder die man zur besseren Strukturierung eines Programms vom Rest des Programms abtrennen will, realisiert man durch Methodenaufrufe. Dazu müssen also Methoden definiert werden. Die Kopfanweisung einer Methodendeklaration kann folgende Form haben:

```
public static TYP name (f1, f2, ...)
```

Dabei gibt `TYP` den Ergebnistyp der Metode an, der mit der `return`-Anweisung erzeugt wird; damit wird die Arbeit der Methode beendet.
Das folgende Beispiel realisiert das sogenannte Hornerschema zur Berechnung des Werts $f(x_0)$ für ein Polynom $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$, dabei wird die Ausklammerung $f(x) = (\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$ genutzt, um das Berechnen von Potenzen zu vermeiden.

```
import java.io.*;
public class horner
{
public static double horn (double[] a, double x0)
{
  int i, n;
  double f;
  n = a.length - 1;
  f = a[n];
  for (i = n-2; i >= 0; i--)
    f = f * x0 + a[i];
  return f;
}

public static void main(String arg[])
{
  double a[] = {1,1,1};          // also x^2+x+1
  double h;
  h = horn(a, 2.0);
  System.out.println(h);
}
```

Falls keine Parameter übergeben werden (müssen), ist eine leere Parameterliste ( ) zu übergeben. Es sind rekursive Aufrufe möglich.
Beim Aufruf eines Unterprogramms wird bei nichtprimitiven Typen direkt auf den Speicherplatz, den die Parameter belegen, zugegriffen (call by reference), d.h. auch die Eingangsparameter können verändert werden (Vorsicht!). Als aktuelle Parameter können nicht nur Variable, sondern auch Ausdrücke übergeben werden (die kann man naturgemäß nicht ändern). Eine sorgfältige Dokumentation aller Parameter ist unbedingt anzuraten.
Eine einfache Sortiermethode:

```
public static void sort(int[] feld)
{
  int i, j, l = feld.length - 1, tausch;
  for (i = 1; i <= l-1; i++)
    for (j = i+1; j <= l; j++)
      if (feld[i] >= feld[j])
      {
        tausch = feld[i];
        feld[i] = feld[j];
        feld[j] = tausch;
      }
}
```

3.6 Programmeinheiten

Die Java-Programmeinheiten sind Klassen. In einer Klasse können Objekte und Methoden vereinbart werden. Ein ausführbares Programm (eine Applikation) enthält eine `main`-Methode, an die Parameter von der Komandozeile übergeben werden können. In jeder Klasse kann (zu Testzwecken) eine `main`-Methode implementiert werden.

```
public class Klasse
{
  public static void main(String arg[])
  {
    int i;
    if (arg.length == -1)
      System.out.println("Eingabe erwartet");
    else
    {
      for (i = 0; i <= arg.length - 1; i++)
        System.out.print(arg[i] + " ");
      System.out.println();
    }
  }
}
```

Methoden für Objekte

```
public class rational
{
  int zaehler, nenner;       // das Objekt rational hat 2 Komponenten
  public int ganzerTeil()
  {
    return this.zaehler / this.nenner;
  }
```

```
}

import rational;
import java.io.*;
public class test
{
  public static void main(String a[])
  {
    rational c = new rational();
    c.zaehler = 5;
    c.nenner = 2;
    System.out.println(c.ganzerTeil());
  }
}
```

Die Superklasse `object` enthält Methoden, die für alle Arten von Objekten zugänglich sind, z.B.
`a.equals(b)` Vergleich, nicht: `a == b`
`b = (TYP)a.clone()` kopiert, nicht: `b = a`
`s = a.toString()` ergibt eine (druckbare) Zeichenkette.

Bei Deklarationen können (gleichnamige) Objekte in übergeordneten Klassen verdeckt oder auch überschrieben werden. Wenn man das vermeiden will, sollte man den Variablen ihren Klassen-Namen aufzwingen.

Wenn eine Klasse nur Variable (sog Instanzmerkmale), aber keine statischen Methoden enthält, so entspricht dies dem Verbunddatentyp aus C oder Pascal (`struct` bzw. `record`).

Objekte werden angelegt, indem eine Variable vom Typ der Klasse deklariert und ihr mit Hilfe des `new`-Operators ein neu erzeugtes Objekt zugewiesen wird:

```
rational c = new rational();
```

Es wird eine neue Instanz der KLasse `rational` angelegt. **Attribute von Klassen, Methoden und Variablen**

`public`: in der Klasse, in abgeleiteten Klassen und beim Aufruf einer Instanz der Klasse verfügbar;
`private`: nur innerhalb der Klasse verfügbar;
`static`: nicht an die Existenz eines konkreten Objekts gebunden (wie oben `ganzerTeil`, `equals` usw.), existiert solange wie die Klasse;
`final`: unveränderbar (Konstanten).

Klassenvariable (Attribut `static`) werden nur einmal angelegt und existieren immer; sie können von jedermann verändert werden (sie sind so etwas wie globale Variable). Die Bezugnahme geschieht mit
`klasse.name`

Nicht-statische Methoden werden beim Erzeugen einer Instanz der Klasse verfügbar:

```
class c {
  public c {};
  public String i() {};
}
```

Aufruf:

```
c nc = new c();
String x;
x = nc.i();
```

Die folgende Methode liefert nicht das, was man erwartet hat:

```
public static void tausch (int i, int j)
{
int h = i; i = j; j = h;
}
```

Hier wird zwar lokal etwas vertauscht, dadurch ändern sich aber die Werte `i, j` nicht.

Wenn die Parameter einer Methode einen nichtprimitiven Typ haben und verändert werden, wirks sich dies global aus. Parameter werden mittels „call by value“ übergeben; hier sind es Adressen, welche auch unverändert bleiben. Aber die Inhalte können verändert werden.

Java importiert automatisch das Paket `java.lang.*`, dazu gehört z.B. die Klasse `Math`, hier finden man z.B. `Math.sqrt(x)`, die `Math` - Methoden haben als Ergebnistyp `double`. Nützlich ist auch `Integer.MAX_VALUE`.

Programmelemente – Zusammenfassung

- Anweisungen (können auch Deklarationen enthalten),
- Blöcke

  ```
  {anw1;
   anw2;
   dekl1;    //nur im Block gueltig
   anw3;
  }
  ```

  Ein Block ist nach außen nur *eine* Anweisung (nach `if(...)` usw. darf nur eine Anweisung stehen, z.B. ein Block).

- Methoden: sie haben Namen, evtl. Parameter und Rückgabewerte; ihre Parameter sind als Referenzen (Zeiger) angelegt (call by reference);
- Klassen: sie enthalten Variable, die den Zustand von Objekten beschreiben, und Methoden, die das Verhalten von Objekten festlegen; Klassen sind schachtelbar: ihre Variablen können den Typ derselben Klasse haben; damit sind rekursive Datenstrukturen (Listen) realisierbar;

Beispiel: Polymome

```
import java.math.*;
import java.io.*;
import java.util.*;
import hgR.*;  // Operationen mit rationalen Zahlen

public class hgP
{
 public hgR co;
 public int ex;
 public hgP next;

public hgP()   // Konstrukteur
{
  this.co = hgR.assign(0);
  this.ex = 0;
  this.next = null;
}

public static boolean iszero(hgP p)
{
  return (!(p instanceof hgP) || hgR.zero(p.co));
}

public static void writep(hgP p)
{
  if(!(p instanceof hgP))
    System.out.print(" NULL ");
  else if (hgR.zero(p.co))
    System.out.print(" zero ");
  while (p instanceof hgP)
  {
    if (hgR.positiv(p.co))
      System.out.print("+");
    hgR.aus(p.co);
    System.out.print("x^" + p.ex);
    p = p.next;
  }
  System.out.print("  ");
}
```

Wenn eine Klasse importiert wurde, reicht der Klassenname zur Dereferenzierung einer Methode / Variablen aus (nicht der ganze Name ist nötig);

- Pakete: Sammlungen von Klassen; jeder Methoden- oder Variablenname besteht aus drei Teilen:
  - Paketname,
  - Klassen- oder Objektname,
  - Methoden- oder Variablenname, z.B.

    `java.lang.Math.sqrt`

  Java weiß natürlich, wo es seine eigenen Pakete zu suchen hat, das obige steht in

  `.../java/classes/java/lang/Math`

  evtl. gibt es unter `java` auch nur eine `zip`-Datei, die gar nicht ausgepackt werden muß (der Compiler weiß, wie er damit umzugehen hat).
- Applikationen (= Programme): Klassen mit `main`-Methoden;
- Applets: werden aus einer HTML-Seite heraus aufgerufen.

Eigene Pakete erstellt man, indem man den beteiligten Klassen sagt, daß sie beteiligt sind:

```
package P;
public class A
{
...
}
```

Dies gehört in die Datei `A.java`, wie ja der Klassenname sagt. Woher kann aber der Compiler eine Information über das Paket `P` erhalten? Es gibt keine Datei mit diesem Namen. Man kann sein Paket aber auch nicht unter `.../classes` unterbringen, denn dort hat man keine Schreibrechte. Also: Die Datei `A.java` gehört ins Verzeichnis

`./P`

Dort wird sie gesucht und gefunden. Pakete nutzt man mit
`import P.*;`
Die Klassen im aktuellen Verzeichnis `.` werden zu einem namenlosen Paket zusammengefaßt; deren Methoden können *ohne* Importe verwendet werden. Das Paket `java.lang` wird automatisch importiert, man hat also automatisch eine Funktion wie `Math.sin` zur Verfügung.

Entwicklung eines Programms

1. Quelltext erstellen, z.B `T.java`,
2. Übersetzen des Quelltexts mittels

   `javac T.java`

ggf. werden Syntaxfehler angezeigt, die zu korrigieren sind. Wenn keine Fehler vorhanden sind, entsteht die Datei

`T.class`

dies ist keine ausführbare Datei (kein Maschinencode), sondern „Java-Bytecode“, der auf allen Plattformen durch einen Java-Interpreter ausgeführt werden kann:

`java T`

3.7 Grafik

Ein Beispiel:

```
import java.awt.*;       // fuer Grafik
import java.awt.event.*; // zum Abschluss

public class sinus extends java.applet.Applet
// Die Applet-Klasse wird erweitert; ihre Methoden stehen hier zur Verfuegung.
{
  public static int    s = 100;
     // 60 Pixel soll die Einheit sein, (nur) hier ist ggf. zu ndern;
     // s soll ein Teiler von 600 sein
  public static double p = 1.0 / s;   // 1 Pixel

  public void paint(Graphics g)
  {
    int ya, y, i,j;
    for (i = 0; i <= 600; i = i+s)               // Koordinatensystem
      g.drawLine(i,0,i,600);
    for (i = 0; i <= 600; i = i+s)
      g.drawLine(0,i,600,i);
    g.setColor(Color.red);
    g.drawLine(0,300,600,300); g.drawLine(300, 0, 300, 600);
    g.setColor(Color.blue);
    ya = (int)(-Math.sin(-300*p)*s) + 300;   // Anfang
    for (i = 1; i<= 600; i++)
    {
      y = (int)(-Math.sin((i-300)*p)*s) + 300;  // verschieben
      if ((y > 0) && (y < 600))          // drin ?
        g.drawLine(i-1,ya,i,y);          // von alt nach neu
      ya = y;                            // weitersetzen
    }
  }

public static void main(String[] arg)
{
```

```
  double w;
  Frame f = new Frame("Sinus");
  f.setBackground(Color.white);
  f.addWindowListener(new WindowAdapter()
   { public void windowClosing(WindowEvent e) {System.exit(0);}} );
  sinus m = new sinus();
  f.setSize(600, 600);   f.show();
}
}
```

3.8 Dateibehandlung

Dies wollen wir uns nur anhand einiger Beispiele (Rezepte) ansehen, ohne auf die Bedeutung der Funktionen einzugehen.

```
import java.io.*;
public class ausgabe       // Guido Krueger
{
public static void main(String arg[])
{
  BufferedWriter f;
  String s;
  if (arg.length == -1)
    System.out.println("Dateiname erwartet");
  else
  {
    try
    {
      f = new BufferedWriter(new FileWriter(arg[0]));
      for (int i = 1; i <= 100; i++)
      {
        s = "Dies ist die " + i + ". Zeile";
        f.write(s);
        f.newLine();
      }
      f.close();
    }
    catch(IOException e)
    {
      System.out.println("Fehler");
    }
  }
}

import java.io.*;
```

```
public class nummer
{
public static void main(String arg[])
{
  LineNumberReader f;
  String line;
  try
  {
    f = new LineNumberReader(new FileReader("nummer.java"));
    while ((line = f.readLine()) != null)
    {
      System.out.print(f.getLineNumber() + ": ");
      System.out.println(line);
    }
    f.close();
  }
  catch(IOException e)
  {
    System.out.println("Fehler");
  }
}
}
```

Zum Schluß noch ein schönes Beispiel:

```
import java.math.BigInteger;
import java.util.*;      // insbes. wird die Klasse Vector importiert,
                         // das ist ein Feld, das wachsen kann
public class factor      // D. Flanagan
{
  protected static Vector table = new Vector();
  static { table.addElement(BigInteger.valueOf(1));

  public static BigInteger factorial(int x)
  {
    if (x < 0)
      throw new IllegalArgumentException("x darf nicht negativ sein");
    for (int size = table.size(); size <= x; size++)
    {
      BigInteger lastfact = (BigInteger)table.elementAt(size - 1);
      BigInteger nextfact = lastfact.multiply(BigInteger.valueOf(size));
      table.addElement(nextfact);
    }
    return (BigInteger)table.elementAt(x);
  }
```

```
public static void main(String[] arg)
{
  for (int i = 1; i <= 50; i++)
    System.out.println(i + "! = " + factorial(i));
}
```

Den Umgang mit Dateien (genauer: die notwendige Behandlung eventueller Ausnahmen) lernt man am Besten anhand von Beispielen, z.B. bei Flanigan, S. 175 - 212.

4 Kommunikation mitttels Computer

Wenn Rechner miteinander verbunden werden, kann man von einem Rechner aus auf die Resourcen des anderen zugreifen: Man kann die auf der anderen Festplatte vorhandenen Daten und Programme nutzen. Die einfachste Verbindung zweier PCs geschieht durch ein Nullmodem, das sind vier Drähte, die bestimmte Anschlüsse der seriellen oder parallen Schnittstellen verbinden; dann braucht man noch etwas Software, die den Datenaustausch organisiert.
In den USA wurde Ende der 60er Jahre begonnen, Computer militärischer Einrichtungen und von Universitäten zu verbinden. Das Internet, ein die ganze Welt umspannendes Netz von ca. 3 Millionen Computern, entstand Mitte der 80er Jahre.

Die einfachste Art der Kommunikation ist das Versenden und Empfangen von Briefen: die elektronische Post, e-mail. Zur Verwendung von e-mail sind drei Dinge notwendig: Der Text eines Briefs muß vorhanden sein. Die Adresse des Empfängers muß bekannt sein. Es muß ein Programm vorhanden sei, das die Post verschickt. Zuerst zur Adresse: Jede Person, die Zutritt zum Internet erlangt, erhält mit seinem Login gleichzeitig eine e-mail-Adresse. Wenn sein Login-Name N lautet, er bei der Institution I arbeitet und im Land L wohnt, könnte seine e-Mail-Adresse

```
n@i.l
```

lauten, z.B. `hgrass@mathematik.hu-berlin.de`. Daß die Nutzernamen innerhalb eines lokalen Netzes eindeutig sind, wird durch die jeweilgen System-Verantwortlichen gewährleistet. Daß sich die verschiedenen Rechner voneinander unterscheiden lassen, ist durch die Vergabe eindeutiger Rechnernummern durch die in jedem Land wirkenden Netzwerk-Informations-Zentren (NIC) zu sichern.
Nun zum Verschicken: Wenn die Datei D den zu versendenden Text enthält, so schreibt man

```
mail n@i.l < d
```

dabei wird dem Programm mail die Datei D als Eingabedatei übergeben. Wenn man nur

```
mail n@i.l
```

schreibt, folgt die Aufforderung

```
Subject:
```

und man soll einen „Betreff“ eingeben. Danach schreibt man seinen Text. Die Eingabedatei für mail ist jetzt die Tastatur. Man beendet eine Datei, indem man das Dateiende-Kennzeichen `<cntrl>+Z` (unter DOS) bzw. `<cntrl>+D` (unter Unix / Linux) eingibt, oder indem man eine Zeile schreibt, die nur aus einem Punkt besteht; letzeres ist am einfachsten. (Es gibt auch mailtools, die etwas mehr Komfort bieten.)
Man kann eingegangene Post lesen, indem man das Programm mail ohne Parameter aufruft. Man erhält zunächst eine Liste der Absender eingegangener Briefe, aus denen man sich einen auswählen kann.
Die herkömmliche gelbe Post wird von e-mail-Nutzern als „snail mail“ bezeichnet, sie hat aber auch einige Vorzüge: Wirklich wichtige Dinge sollte man nicht ausschießlich per e-mail verschicken; aus diesem oder jenen Grund kann elektronische Post auch verlorengehen. Man soll sich also bestätigen lassen, daß die Nachricht angekommen ist. Auf der anderen Seite soll man wirklich vertrauliche Dinge auch nicht, zumindest nicht unverschlüsselt, dem Internet anvertrauen (fast jeder kann alles lesen). Allerdings denken die entsprechenden staatlichen Stellen intensiv darüber nach, wie es zu verhindern ist, daß unkontrolliert verschlüsselte Botschaften versandt werden.

Mit dem Programm `telnet`, dem ersten Dienst, der im Internet eingerichtet wurde, kann man auf einem fremdem Rechner so arbeiten, als säße man davor. Man schreibt

```
telnet rechnername
```

und wird vom fremden Rechner zum Einloggen aufgefordert. Zum Einloggen muß man natürlich berechtigt sein, aber wenn man das nicht wäre, wüßte man sicher gar nicht den Namen des Rechners, den man nutzen will. Es ist allerdings zu beachten, daß das Paßwort unverschlüsselt über die Leitung geschickt wird. Bei Verwendung des Dienstes `ssh` (security shell) ist diese Sicherheitslücke geschlossen, jedoch muß dieser Dienst auf beiden beteiligten Rechnern installiert sein, was man nicht ohne weiteres voraussetzen kann.

Ein weiterer Dienst ist das File Transfer Protocol `ftp`. Es gibt an vielen Universitäten spezielle Rechner, die nichts anderes tun, als Daten für den Austausch bereitzuhalten. Dies sind die FTP-Server. Auf diese Rechner darf jeder zugreifen, man schreibt:

```
ftp rechnername
```

und wird nach seinem Namen gefragt

```
username:
```

Es ist üblich, daß ein Login mit dem Namen `anonymous` oder einfach `ftp` eingerichtet ist, mit diesem loggt man sich ein. Danach wird ein „Paßwort“ abgefragt, das keins ist. Man wird gebeten, als Paßwort seine vollständige e-mail-Adresse anzugeben. Dann folgt der Prompt

```
ftp>
```

Normalerweise weiß man, in welchem Verzeichnis die Daten liegen, die man sich holen will. Wenn nicht, so muß man den Verzeichnisbaum durchstöbern. Mit dem Kommando

```
ls
```

schaut man sich Inhaltsverzeichnisse an, mit

```
cd
```

wechselt man Verzeichnisse. Mit

```
get filename
```

oder

```
put filename
```

holt man sich eine Datei bzw. schreibt eine Datei (falls man dies darf, dies ist nicht automatisch erlaubt, ggf. muß man sich mit dem Administrator des Servers in Verbindung setzen). Will man mehrere Dateien lesen oder schreiben, so gibt man am besten zunächst das Kommando

```
prompt
```

das schaltet Einzelrückfragen ab, danach z.B.

```
mget *.dat
```

oder

```
mput a*
```

Dabei werden ASCII-Dateien übertragen. Will man binäre Dateien, z.B. gepackte Programme, übertragen, so sendet man vorher das Kommando

```
bin
```

Mit

```
quit
```

kappt man die Verbindung.

Mit den Diensten Gopher und WWW wurde ein bequemer Zugriff auf weltweit verstreute Informationen möglich. Hierzu gab es zunächst zeilenorientierte „Web-browser“, also „Netz-Stöberer“; das World Wide Web wurde 1993/94 bekannt, als mit dem Browser `mosaic` eine grafische Oberfläche geboten wurde. Wenn die Texte im Format Hypertext Markup Language (HTML) erstellt wurden, so kann man sich durch einen Mausklick auf markierte Worte automatisch auf andere Rechner/Dateien weiterleiten lassen. Dieses System wurde von Marc Andreesen entwickelt, er war damals Student, später gründete er die Firma Netscape, deren gleichnamiger Browser `mosaic` so gut wie verdrängt hat.

In die Gestaltung ihrer Internet-Seiten verwenden die Autoren immer aufwendigere Techniken, bunte, bewegte Bilder usw., oft einfach, um aufzufallen. Daraus resultieren teilweise erhebliche Ladezeiten. Einige Browser zeigen den aktuellen Datendurchsatz beim Laden an. Wenn dieser bei 1 ... 2 KByte / s liegt, kann man froh sein, bei 150 Byte / s wird man ungeduldig. Das Wort "stalled" heißt: abgewürgt.
Für den Hausgebrauch (wenn also beim Surfen Telefonkosten anfallen) ist die Verwendung der frei erhältlichen Browser `lynx` oder `arena` vorteilhaft, diese übertragen keine Bilder und Farben und unterstützen keine Maus-Operationen; den Links folgt man mit den Pfeil-Tasten.
Wenn es einem einmal gelungen ist, ins Netz einzudringen (dazu muß man nur eine Adresse kennen), so wird man durch freundliche vorhandene Insassen um die ganze Welt geleitet. Auf die Heimatseite desInstituts für Mathematik gelangt man durch Eingabe der URL(uniform resource locator)

```
http://www.mathematik.hu-berlin.de/
```

Dort kann man sich das Vorlesungsverzeichnis ansehen, zur Informatik, dem Rechenzentrum oder zu anderen mathematischen Instituten in Berlin bzw. Deutschland durchhangeln oder in den von Mitarbeitern und Studenten gestalteten „persönlichen Seiten" nachschauen, worauf diese hinweisen wollen. Da ist man schnell in der Gemäldegalerie von Bill Gates gelandet.
Zur Gestaltung eigener Web-Seiten ist es zweckmäßig, zuerst mal eine Seite von einem Bekannten, die in HTML geschreiben ist, zu kopieren (der Bekannte ist auf die gleiche Weise zu seiner Vorlage gekommen) und sich als ASCII-Text anzusehen. Da erfährt man schon viel über die Struktur dieses Formats. Nun ändert man die persönlichen Angaben einfach ab und speichert die Datei unter einem vom SysOrg vorgegebenen Namen, oft `Wellcome.html` oder `index.html` in einem ebenfalls vorgegebenen Verzeichnis, in dem alle Nutzern Leserechte haben, ab; das Verzeichnis könnte z.B. `public_html` heißen.
(Unter DOS ist die Dateinamenserweiterung `.htm` zu verwenden.)
Eine Einführung in HTML wird in dem „Buch" *HTML-Einführung* von H. Partl gegeben, das im Web u.a. unter der URL

```
http://www.mathematik.hu-berlin.de/~richter/hein.html
```

zu finden ist, gegeben.
Eine Einführung ins Internet findet man in dem Buch von M. Scheller u.a., das ebenfalls im Web lesbar ist; fragen Sie unter `scheller@ask.uni-karlsruhe.de` an.

Wenn man an einem Rechner sitzt und auf einem anderen etwas editieren will, muß man dazu natürlich berechtigt sein. Man kann aber evtl. seinen favorisierten Editor nicht benutzen, weil der fremde Rechner den gar nicht kennt oder nicht weiß, wohin der denn die Daten schicken soll (der fremde Rechner weiß nichts über Sie). Hier hilft der Editor `vi`, der auf jedem Unix-Rechner installiert ist. Der Aufruf erfolgt mittels

```
vi dateiname
```

Mit dem Kommando `:set nu` kann man sich Zeilennummern anzeigen lassen. Nun kann man sich mit den Kursor-Tasten im Text bewegen, wenn das nicht geht, helfen

die Tasten `h, j, k, l` und die Return-Taste. Das Zeichen unter dem Kursor streicht man mit `x`, man ersetzt es mit `r neues-Zeichen`. Die aktuelle Zeile streicht man mit `dd`.
Um etwas in den Text einfügen zu können, muß man sich mittels `i` in den Insert-Modus begeben oder mit `o` bzw. `O` eine Leerzeile unter bzw. über der aktuellen einfügen.

Nun kann man schreiben. Den Insert-Modus verläßt man mit der Escape-Taste. Zum Sichern schreibt man `:wq`.

Das waren nur die allereinfachsten Kommandos, natürlich gibt es auch über `vi` ganze Bücher.

5 LaTeX

In diesem Abschnitt soll kurz ein Texterstellungssystem vorgestellt werden, das sich besonders zum Schreiben mathematischer Texte eignet.
Heute ist auf PCs das Textverarbeitungssystem Word weitverbreitet. Hier hat man Menüs, an denen man viele Verarbeitungseigenschaften (Schriftgrößen usw.) einstellen kann, und man hat eine Schreibfläche, auf der der Text so dargestellt wird, wie er später gedruckt aussieht. Derartige Textverarbeitungssysteme machen aus dem PC eine komfortable Schreibmaschine. Die Datei, die den gespeicherten Text enthält, enthält außerdem zahlreiche Formatierungsinformationen, oft in Form nichtdruckbarer Zeichen. Eine derartige Datei sollte man nie „von Hand“ verändern.

Demgegenüber ist TeX ein Programm, das einen Text als Eingabe erwartet und daraus ein Ergebnis erarbeitet, das alle Informationen enthält, um fertige Druckseiten zu erstellen. TeX erfüllt also die Aufgabe eines Schriftsetzers; es ist als Programmiersprache eines Satzautomaten aufzufassen. TeX wurde 1978 bis 1982 von Donald Knuth an der Stanford University entwickelt. Einen weiten Anwenderkreis hat es durch das von Leslie Lamport geschaffene Werkzeug LaTeX erhalten.
Es versteht sich, daß dieser Text mittels LaTeX (genauer: `glatex` von `emtex` erstellt wurde.
Die Eingabedatei – eine reine ASCII-Datei – stellt man mit Hilfe eines Editors her. Ein „Wort“ ist eine Zeichenkette, die kein Leerzeichen und kein Zeilenende-Kennzeichen enthält (dies sind Trennzeichen, die auf die späteren Wortabstände keinen Einfluß haben). Ein Absatz wird durch eine Leerzeile beendet, Zeilen- und Seitenumbrüche werden automatisch durchgeführt. Innerhalb des Textes können Befehle stehen, die entweder mit einem Backslash `\` beginnen oder ein verbotenes Zeichen darstellen; verbotene Zeichen, die also eine besondere Bedeutung haben, sind

`#, $, &, ~, _, ^, %, {, }.`
Wenn ein `#` im Text nötig ist, so muß man `\#` schreiben.
Befehle haben eventuell Argumente:

```
\befehl[optional]{zwingend}
```

Im Vorspann werden globale Eigenschaften des Texts festgelegt:

- Papierformat,
- Textbreite und -höhe,
- Seitenköpfe, Numerierung.

Unumgänglich sind die folgenden Angaben

```
\documentstyle[Optionen]{Stiltyp} bzw. \documentclass...
\begin{document}
```

Dann folgt der Text. Am Ende steht

```
\end{document}
```

Der Name der Eingabedatei muß auf `.tex` enden, z.B. `text.tex`, der Aufruf `latex text` erzeugt die Datei `text.dvi`, die den formatierten Text in druckerunabhängiger Form enthält. DVI steht für „device independent“, und das ist ernst zu nehmen: man kann eine DVI-Datei am PC erzeugen und an Unix-Rechner anschauen. Diese Datei wird schließlich von einem Treiber (z.B. `dvips`) zu einer druckbaren Datei weiterverarbeitet.
Eingestellte Optionen gelten innerhalb einer bestimmten Umgebung. Eine Umgebung beginnt mit

```
\begin{umg}
```

und ended mit

```
\end{umg}.
```

Die Wirkung von Änderungsbefehlen (Zeichengröße, Schriftart usw.) endet beim aufhebenden Änderungsbefehl oder am Umgebungsende. Beispiele für Umgebungsnamen sind `center, quote` (beidseitig einrücken), `flushleft, flushright`. Es gibt auch namenlose Umgebungen {}, z.B. `{\bf fetter Druck}` erzeugt **fetten Druck**.
Innerhalb der `verbatim`-Umgebung werden alle Zeichen (auch Leerzeichen und Zeilenumbrüche) im `tt`-Format genauso gedruckt, wie sie geschrieben wurden.

5.1 Maßangaben

Als Maßeinheiten können Zentimeter `cm`, Millimeter `mm`, Zoll `in`, Punkte `pt`, Picas `pc`, die Breite eines Gedankenstrichs `em` und die Höhe des kleinen x `ex` verwendet werden. Ein Zoll sind 72,27 Punkte, ein Pica sind 12 Punkte. Beispiele:

```
\setlength{\textwidth}{12.5cm}
\setlength{\parskip}{1ex plus0.5cm minus0.2cm}
```

`\parskip` bezeichnet den Absatzabstand, mit `plus` und `minus` kann man Dehn- und Schrumpfwerte angeben, die (wenn möglich) Anwendung finden; damit hat man „elastische Maße“. Besonders elastisch ist das Maß `\fill`, z.B. vergrößert man Abstände mit

```
\hspace{\fill}
```

Der Parameter von `hspace` kann auch negativ sein.
Sonderzeichen in speziellen Sprachen gibt es auch:

```
\pounds, \OE, \'{o}, "A, "s, \today
```

Dies stellt £, Œ, ó, Ä, ß, 10. Januar 2001 dar.

5.2 Stile

Man kann sich aus vorgefertigten Dateien Standard-Einstellungen laden:

```
\documentstyle[option, ...]{stil}
```

Optionen beziehen sich jeweils auf eine `.sty`-Datei, z.B.
`11pt, 12pt, twoside, twocolumn, bezier, german, a4`.
Als Stile gibt es `book, article, report, letter`. Ebenfalls im Vorspann wird der Seitenstil vereinbart:

```
\pagestyle{stil}
```

Als Stile kommen in Frage: `empty` (keine Seitennummern), `plain` (Seitennummern), `headings` (lebende Seitenüberschriften). Den Stil der aktuellen Seite kann man abweichend festlegen: `\thispagestylestil`. Abstände sind z.B. `\baselineskip` (Zeilenabstand), `\parskip` (Absatzabstand) und `\parindent` (die Größe des Einzuge der ersten Zeile eines Absatzes). z.B.

```
\setlength{\parskip}{0cm}
```

5.3 Untergliederungen

```
\chapter, \section, \subsection
```

unterteilt in Abschnitte, `\chapter` gibt es nur in Büchern. Z.B. gibt man dem Abschnitt „Gleichungssysteme“ die Seitenüberschrift „Systeme“ durch

```
\section[Systeme]{Gleichungssysteme}
```

Die einzelnen Abschnitte werden automatisch durchnummeriert. Das kann man auch selbst beeinflussen:

```
\setcounter{chapter}{-1}
```

Mittels

```
\tableofcontents
```

erstellt man ein Inhaltsverzeichnis. Wenn dies am Anfang des Texts stehen soll, muß man LaTeX zweimal denselben Text bearbeiten lassen, denn es wird stets die bisher vorhandene Information über das Inhaltsverzeichnis benutzt.

5.4 Schriftarten

Folgende Schriftarten stehen zur Verfügung:
`\rm` (Roman, 10pt, dies ist der Standard), `\em` (emphasize, kursiv), `\bf` (fett), `\tt` (Schreibmaschine), `\sl` (slanted, geneigt), `\sc` (small capitals), `\sf` (sans serif).
Dieser Text *ist jeweils* **(immer zwei** `Worte weise`**)** *mit den* VORHANDENEN SCHRIFTARTEN `gesetzt worden`.

Die Größe der Buchstaben ist eine der folgenden:

`\tiny, \small, \normal, \large, \Large, \LARGE, \huge, \Huge.`

5.5 Aufzählungen

```
\begin{itemize}
\item erstens
\item zweitens
\end{itemize}
```

ergibt

- erstens
- zweitens

Außerdem gibt es `enumerate` und `description`, dabei werden die Items numeriert bzw. an Stichworten festgemacht. All das kann man auch verschachteln. Um die Markierung zu ändern, schreibt man z.B. in den Vorspann:

```
\renewcommand{\labelitemiii}{+}}
```

Dabei wird in der dritten Schachtelungsstufe als Marke ein + verwendet.

5.6 Regelsätze

Hierunter versteht man die in mathematischen Texten übliche gleichartige Gestaltung der Form

Satz 5.1 (Gauß) *Dies ist der Satz von Gauß.*

Satz 5.2 (Fermat) *Dies ist der Satz von Fermat.*

Man erreicht dies durch die Vereinbarung

```
\newtheorem{satz}{Satz}[chapter]
```

und den Aufruf

```
\begin{satz}[Gau"s]
 ...
\end{satz}
```

Dabei ist `satz` der Bezeichner und `Satz` der zu setzende Titel; die Angabe von `chapter` in der Vereinbarung hat zur Folge, daß Sätze innerhalb eines Kapitels durchnumeriert werden. Wenn auch Lemmata, Folgerungen usw. gemeinsam durchnumeriert werden sollen, so kann man

```
\newtheorem{lemma}[satz]{Lemma}
\newtheorem{folg}[satz]{Folgerung}
```

vereinbaren.

5.7 Kästen

Durch

```
\framebox[4cm][l]{Text}
```

wird ein Kasten erzeugt, der linksbündigen Text Text enthält.
Wenn anstelle von der Positionierungsangabe `l` ein `r` steht, wird der Text rechtbündig angebracht. rechter Text Ohne Angabe wird er zentriert.

5.8 Teilseiten (minipages)

Durch eine Vereinbarung wie

```
\begin{minipage}[pos]{breite} .... \end{minipage}
```

wobei `pos` die Ausrichtung an der aktuellen Zeile bestimmt und die Werte `b, t, c` annehmen kann (bottom, top, centered), erstellt man einen Textteil, der logisch wie ein Wort behandelt wird; man kann also mehrere Teilseiten nebeneinander setzen (man nennt dies auch mehrspaltigen Satz).
Ich mußte jedoch die Erfahrung machen, daß innerhalb von minipages die ansonsten automatisch durchgeführte Silbentrennung nicht funktionierte; hier muß man selbst Hand anlegen: wenn `\-` in die Silbenfugen eingefügt wird, entsteht dort eine mögliche Trennungsfuge.

5.9 Tabellen

```
\begin{Tab-Typ}[pos]{spaltenformat} ... \end{Tab-Typ}
```

Der `Tab-typ` kann `tabular` oder `array` sein, für `pos` kann `t` oder `b` stehen (Ausrichtung an aktueller Zeile), als `spaltenformat` sind möglich:

`l, r, c`	:	links- oder rechtsbündig bzw. zentriert
`|`	:	senkrechter Strich
`||`	:	zwei Striche
`*{5}{|c|}`		ergibt `|c|c|c|c|c|`

Die Zeilen einer Tabelle sind durch `\\` zu trennen, der jeweilige Spaltenanfang wird durch `&` markiert (außer 1. Spalte; die Spalten werden richtig untereinandergesetzt), mit `\\ \hline` wird eine waagerechte Linie eingefügt.

5.10 Mathematische Formeln

1. In mathematischen Texten werden Variable stets in einer anderen als der Grundschrift gesetzt, normalerweise kursiv. Die Bezeichner von Standardfunktionen werden jedoch steil gesetzt.

2. Es gibt eine Vielzahl vordefinierter mathematischer Symbole, wie $\sum$, $\int$, $\in$, $\times$, $\longrightarrow$ usw, die nur in einer Mathematik-Umgebung vorkommen dürfen. Es gibt zwei derartige Umgebungen: eine für Formeln, die in der laufenden Zeile vorkommen, und eine für „abgesetzte“ Formeln, die einen neuen Absatz eröffnen und zentriert gesetzt werden. Als Umgebungsgrenzen können jeweils ein oder zwei Dollarzeichen genommen werden, z.B.:

   ```
   Es gilt $ \sum _{i=1} ^n i = \frac{(n+1)n}{2} $
   ```

   ergibt

   Es gilt $\sum_{i=1}^{n} i = \frac{(n+1)n}{2}$,

   während

   ```
   $$ \sum _{i=1} ^n i = \frac{(n+1)n}{2} $$
   ```

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

   ergibt.

3. Wir stellen die einige Möglichkeiten zusammen, für weitere möge man in entsprechenden Tabellen nachsehen.

Exponenten	`x^2, \; x^{a+b}`	$x^2,\ x^{a+b}$
Indizes	`x_2, \; x_{a+b}`	$x_2,\ x_{a+b}$
Brüche	`\frac{c}{a+b}`	$\frac{c}{a+b}$
Wurzeln	`sqrt{x}, \; sqrt[n]{x}`	$\sqrt{x},\ \sqrt[n]{x}$
Summe	`\sum`	$\sum$
Integral	`\int`	$\int$
Punkte	`\ldots \; \cdots \; \vdots \; \ddots`	$\ldots\ \cdots\ \vdots\ \ddots$
Funktionen	`\exp, \; \lim`	exp, lim
Striche etc.	`\overline{xxx}, \; \underbrace{yyy}_n`	$\overline{xxx},\ \underbrace{yyy}_{n}$
große Klammern	`\left[\matrix{1 & 2 \cr 3 &4} \right]`	$\left[\matrix{1 & 2 \cr 3 &4}\right]$

5.11 Einfache Zeichnungen

Eine Bildumgebung wird durch

```
\begin{picture}(breite, hoehe)
```

eingerichtet; dabei bezeichnen `breite` und `hoehe` die Größe des Bildes, das an der aktuellen Stelle eingefügt werden soll. Die Maßzahlen beziehen auf die z.B. durch

```
\setlength{\unitlength}{3cm}
```

festgelegte Längeneinheit. Man stelle sich nun ein Koordinatensytem vor, dessen Ursprung in der linken unteren Ecke des Bildes liegt. Mit

```
\put(x,y){objekt}
```

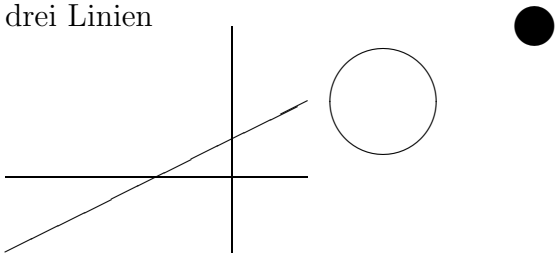
positioniert man an der Stelle (x,y) ein Objekt, dabei ist es zulässig, daß x und y beliebige Werte haben; es wird auch außerhalb des Bildes gezeichnet.
Einen Text bringt man z.B. so an:

```
\put(x,y){text}
\put(x,y){\makebox(0,0){text}}
```

Aber eigenlich soll ja etwas gezeichnet werden.

```
\setlength{\unitlength}{1cm}
\begin{picture}(4,4)
\put(0,1){\line(1,0){4}}
\put(3,0){\line(0,1){3}}
\put(0,0){\line(2,1){4}}
\put(4,4){drei Linien}
\end{picture}
```

drei Linien



ergibt

Eine Linie zieht man mit dem Befehl

```
\line(x,y){proj}
```

Dabei ist y/x der Tangens des Anstiegswinkels, für `x` und `y` sind die Werte 0, 1, 2, 3, 4, 5, 6 zulässig, sie müssen zudem teilerfremd sein. Der Wert `proj` gibt bei senkrechten Linien die Länge, sonst die Länge der Projektion auf die x-Achse an. Analog kann man mit

```
\vector(x,y){proj}
```

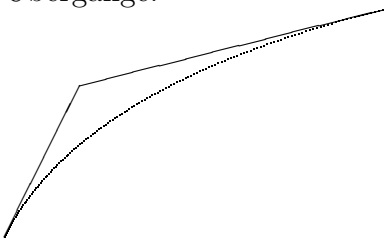
einen Pfeil zeichnen, hier sind als Abszissen Werte zwischen 0 und 4 zugelassen. Mit

```
\circle{durchm},  \circle*{durchm}
```

kann man kleine Kreise bzw. Kreisflächen (bis 1/2 Zoll Durchmesser) zeichnen. Größere Kreise zeichnet man am besten mit Hilfe des Bezier-Stils: Mit

```
\bezier{punktzahl}(x1,y1)(x2,y2)(x3,y3)
```

zeichnet man eine Parabel durch die Punkte P1 und P3, deren entsprechende Tangenten sich im Punkt P2 schneiden; durch Zusammensetzen solcher Kurven erhält man glatte Übergänge.



```
\begin{picture}(5,3)
\bezier{150}(0,0)(1,2)(5,3)
\thinlines
\put(0,0){\line(1,2){1}}
\put(1,2){\line(4,1){4}}
\end{picture}
```

Bilder können auch verschachtelt werden.

5.12 Eigene Befehle

Häufig wiederholte Befehlsfolgen möchte man vielleicht zu einem einzigen neuen Befehl zusammenfassen, wobei evtl. noch Parameter übergeben werden sollen. Die folgenden Konstrukte schaffen neue Befehle bzw. überschreiben alte:

```
\newcommand{\name}[parameterzahl]}{definition}
\renewcommand{\name}[parameterzahl]}{definition}
```

Die Parameteranzahl kann zwischen 0 und 9 liegen. Der Aufruf erfolgt durch

```
\name{...}
```

Beispiele:

```
\newcommand{\xvecnmath}{(x_1, \ldots , x_n)}
\newcommand{\xvecntext}{$(x_1, \ldots , x_n)$}
```

Der Aufruf von `\xvectmath` ist nur im mathememathischen Modus möglich, der von `xvecttext` nur im Textmodus, da eben Indizes „_“ in Texten nicht erlaubt sind. Als Trick für beide Modi kann man

```
\newcommand{\xvecnallg}{\mbox{$(x_1, \ldots , x_n)$}}
```

vereinbaren, dies ist in beiden Modi legal und liefert $(x_1, \ldots, x_n)$ bzw. $(x_1, \ldots, x_n)$. Wenn man aber $(y_k, \ldots, y_l)$ erzeugen will, so kann man Parameter übergeben:

```
\newcommand{\yvecallg}[3]{\mbox{$(#1_#2, \ldots , #1_3)$}}
```

Der Aufruf hierfür würde

```
\yvecallg{y}{k}{l}
```

lauten. Falls jeder Parameter nur *ein* Zeichen enthält, kann man auch `\yvecallgykl` schreiben, das ist (im Quelltext) aber schwer lesbar.

Eigene Definitionen von Befehlen, die man auch andernorts noch verwenden möchte, schreibt man am besten in eine eigene Start-Datei, die man am Anfang des Dokuments mit dem Befehl

```
\input eingabe
```

einliest.

Mittels % leitet man einen Kommentar ein, der bis zum Zeilenende geht.
`\symbol{60}` ergibt das Kleinerzeichen, `\sim` ist eine Tilde; es gibt auch `\smile` ...

Zum Abschluß füge ich meine Standard-Anfangsdatei an. Überlegen Sie, warum die angegebenen Definitionen gewählt wurden und was sie bewirken.

```
\def\Bbb#1{{\bf #1}}
\def\square{\hfill \hbox{\vrule\vbox{\hrule\phantom{o}\hrule}\vrule}}
\documentstyle[german,bezier,12pt]{book}
\textwidth15.5cm
\textheight23cm
\oddsidemargin0mm
\evensidemargin-4.5mm
\topmargin-10mm
\pagestyle{headings}
\markboth{l}{r}
\setlength{\parindent}{0pt}
\setlength{\unitlength}{1cm}
\makeindex
\begin{document}
\newtheorem{satz}{Satz}[chapter]
\newtheorem{lemma}[satz]{Lemma}
\newtheorem{folg}[satz]{Folgerung}
\newcommand{\p}{\par \noindent}
\newcommand{\m}{\par \medskip \noindent}
\newcommand{\diag}[8]
{
$$
\begin{array}{clclc}
  & #1 & \stackrel{\displaystyle #2}{\longrightarrow} & #3 \\
#4 & \Big\downarrow & & \Big\downarrow & #5 \\
  & #6 & \stackrel{\displaystyle #7}{\longrightarrow} & #8
\end{array}
$$
}
\newcommand{\betq}[1]{\left| #1 \right| ^2}
\newcommand{\bet}[1]{\left| #1 \right|}
\newcommand{\pr}[2]{\langle #1, #2 \rangle}
\newcommand{\bi}{\begin{itemize}}
\newcommand{\jj}{\item}
\newcommand{\ei}{\end{itemize}}
\newcommand{\be}{\begin{enumerate}}
\newcommand{\ee}{\end{enumerate}}
\newcommand{\bs}{\begin{satz}}
\newcommand{\es}{\end{satz}}
\newcommand{\bl}{\begin{lemma}}
\newcommand{\el}{\end{lemma}}
\newcommand{\bfo}{\begin{folg}}
\newcommand{\efo}{\end{folg}}
\newcommand{\de}{{\bf Definition: }}
\newcommand{\Pa}{\left\| }
```

```
\newcommand{\ap}{\right\| }
\newcommand{\eps}{\; \epsilon \;}
\newcommand{\la}{\longrightarrow}
\newcommand{\kreis}
{
    \bezier{100}(1.707,1.707)(2,1.414)(2,1)
    \bezier{100}(1,2)(1.414,2)(1.707,1.707)
    \bezier{100}(0.293,0.293)(0,0.586)(0,1)
    \bezier{100}(1,0)(0.586,0)(0.293,0.293)
    \bezier{100}(1.707,0.293)(2,0.586)(2,1)
    \bezier{100}(1,0)(1.414,0)(1.707,0.293)
    \bezier{100}(0.293,1.707)(0,1.414)(0,1)
    \bezier{100}(1,2)(0.586,2)(0.293,1.707)
}
```

6 Komplexität

Die Effizienz von Algorithmen mißt man an der verbrauchten Zeit und dem belegten Speicherplatz. Wir behandeln hier die Zeit-Komplexität.
Wenn verschiedenen Implementierungen (verschiedene Algorithmen), die ein und dieselbe Funktion realisieren, mit denselben Eingabedaten und auf denselben Rechner verglichen werden sollen, genügt die Zeitmessung mit der eingebauten Rechneruhr, z. B. `t = B.zeit(t)`.
Wenn vom Rechner oder, allgemeiner, vom Maschinenmodell abstrahiert werden soll, die Daten aber gleich sind, so kann man die Operationen zählen; der konkrete Zeitbedarf hängt von den Ausführungszeiten auf dem konkreten Rechner ab.
Wenn auch von den Eingabedaten abstrahiert werden soll, so berechnet man die Zahl der Operationen in Abhängigkeit von der Anzahl der Eingabedaten.
Wenn ein Algorithmus für die Verarbeitung von n Daten $10 \cdot n \log n$ Rechenschritte benötigt und ein anderer braucht $n^2 + 10n$, welcher ist dann schneller? Für $n = 10$ braucht der erste etwa 400 Schritte, der zweite nur 200. Aber für $n = 100$ braucht der erste 8000 Schritte und der zweite 10000 und für immer größere Anzahlen schneidet der erste Algorithmus immer besser ab. Wenn es um die Effizienzbestimmung bei großen Datenmengen geht, so nimmt man eine asymptotische Abschätzung des Aufwandes vor. Dabei sieht man von konstanten Faktoren ab: zwei Algorithmen, die Rechenzeiten von einem bzw. zehn Jahren benötigen, sind gleichermaßen inakzeptabel.

6.1 Effektives und uneffektives Rechnen

6.1.1 Der Kreis

Die Aufgabe besteht darin, die Koordinaten der Punkte des Einheitskreises zu berechnen, z.B. um Kreise zu zeichnen.

1. Methode: naiv

Für $\alpha = 0,\ 1, \ldots, 359$ berechne

$$x(\alpha) = \cos(2\pi \cdot \alpha/360),$$

$$y(\alpha) = \sin(2\pi \cdot \alpha/360).$$

Dafür braucht man 720 Funktionsaufrufe.

2. Methode: effektiver

Setze $\beta = 2\pi/360,\ CB = \cos\beta,\ SB = \sin\beta$. Wenn $x(\alpha),\ y(\alpha)$ schon bekannt sind, so ist

$$x(\alpha+1) = \cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta,$$

$$y(\alpha+1) = \sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta.$$

Also:

$x(0) := 1,\ y(0) := 0,$
Für $\alpha = 1, \ldots, 359$:
$CA := x(\alpha),\ SA := y(\alpha),$
$x(\alpha+1) := CA \cdot CB - SA \cdot SB,$
$y(\alpha+1) := SA \cdot CB + CA \cdot SB.$

In der Schleife findet gar kein Funktionsaufruf mehr statt, aber die Rundungsfehler pflanzen sich fort.

3. Methode: Beachten von Symmetrien

Wir lassen den Winkel α nur von 1 bis 45 laufen und weisen neben (x, y) auch $(\pm x, \pm y)$ und $(\pm y, \pm x)$ die entsprechenden Werte zu. Man sollte überprüfen, ob man wirklich nur 1/8 der Zeit dafür braucht.

4. Methode: Tabellieren

Wenn viele Kreise zu behandeln sind (verschobene, mit verschiedenen Radien), so kann man die Daten des Einheitskreises tabellieren und neue Daten daraus berechnen. Bei kleinen Kreisen braucht man evtl. nur jeden 10. Punkt anzufassen.

5. Ganzzahlige Kreiskoordinaten

Wenn Punkte auf den Bildschirm gezeichnet werden sollen, so braucht man ganzzahlige Koordinaten; man kann also im Integer-Bereich rechnen, das geht schnell.
Im Bereich zwischen 0 und 45 Grad ist der obere Nachbarpunkt des Kreispunkts (x, y) entweder der Punkt $(x-1, y+1)$ oder $(x, y+1)$. Wir prüfen also, welcher der Werte

$$|\ (x-1)^2 + (y+1)^2 - r^2\ |$$

und

$$|\ x^2 + (y+1)^2 - r^2\ |$$

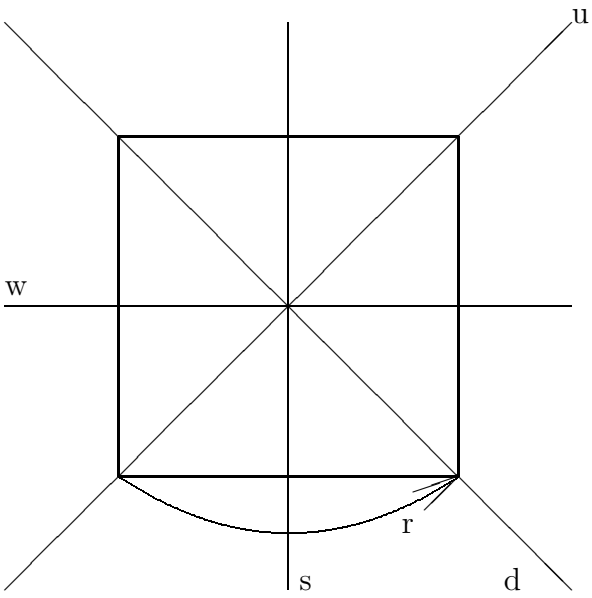
der kleinere ist. Wir berechnen $x(y)$ in Abhängigkeit von y:

$x[0] := r$
für $y = 1, \ldots, \frac{1}{\sqrt{2}}r$ berechne:
$A = (x[y-1])^2 + y^2 - r^2$

wenn $abs(A) < abs(A - 2x[y-1] + 1)$ ist, so
setze $x[y] = x[y-1]$,
sonst $x[y] = x[y-1] - 1$.

Ellipsenpunkte berechnet man, indem man die Koordinatenpunkte des Einheitskreises geeignet mit Faktoren multiplipliziert.

Nochmals Symmetrien: Die Diedergruppe D_4



Ein Quadrat hat 8 Symmetrien: die Spiegelungen an den Achsen w, s, d, u und die Drehungen r^0, r, r^2, r^3 um Vielfache von 90^o. Die Nacheinanderausführung zweier solcher Symmetrien ist wieder eine Symmetrie, sie sind bijektive Abbildungen und für die Komposition von Abbildungen gilt das Assoziativgesetz; die Symmetrien bilden die sog. Diedergruppe D_4.
Es gilt

$$D_4 = \{e, r, r^2, r^3, s, rs = u, r^2s = w, r^3s = d\},$$

diese Gruppe wird also durch die Elemente r, s erzeugt (wobei wir anstelle von s irgendeine Spiegelung einsetzen dürfen). Durch die Rechenregel

$$sr = r^{-1}s$$

sind bereits alle Produkte festgelegt.
Nun ist aber auch jede Drehung ein Produkt zweier (geeigneter) Spiegelungen, etwa

$$r = wd.$$

Wir denken wieder an ein Koordinatensystem und wollen wissen, in welchen Punkt (i, j) bei der Drehung r übergeht. Hier haben wir die Spiegelungen w, d bevorzugt, da diese Operationen am einfachsten zu implementieren sind (der Koordinatenursprung ist links oben, das Quadrat mit der Seitenlänge n soll um seinen Mittelpunkt gedreht werden):

$$d : (i, j) \mapsto (j, i),$$

$$w : (i, j) \mapsto (n - i, j),$$

also

$$r = wd : (i, j) \mapsto (j, i) \mapsto (n - j, i).$$

6.1.2 Der größte gemeinsame Teiler

Den ggT zweier natürlicher Zahlen a, b berechnet man mit Hilfe des Euklidischen Algorithmus:

```
r = 1;
while (r != 0)
{
  r = a % b;
  a = b;
  b = r;
}
return a;
```

Wenn $ggT(a, b) = d$ ist, so gibt es ganze Zahlen u, v mit

$$d = u \cdot a + v \cdot b,$$

die Zahl d ist die betragsmäßig kleinste (von Null verschiedene) Zahl, die sich als Vielfachsumme von a und b darstellen läßt.
Beim Lösen diophantischer Gleichungen ist es hilfreich, eine derartige Darstellung von d zu kennen. Man kann eine solche Darstellung erhalten, wenn man die Schritte des Euklidischen Algorithmus rückwärts verfolgt.
Wir können aber einen der Faktoren auch gleich mitberechnen: Wir setzen

$$r_0 = a; r_1 = b; t_0 = 0; t_1 = 1;$$

es sei

$$r_0 = q_1 r_1 + r_2.$$

Im k-ten Schritt sei

$$r_{k-1} = q_k r_k + r_{k+1},$$

$$t_{k-1} = q_k t_k + t_{k+1},$$

wobei wir das jeweilige q_k in der ersten Zeile berechnen und in der zweiten Zeile zur Berechnung von t_{k+1} verwenden.
Wir setzen nun voraus, daß der Term $r_k - bt_k$ durch die Zahl a teilbar ist; der Induktionsanfang

$$1 \cdot a - b \cdot 0 = a,$$

$$1 \cdot b - b \cdot 1 = 0$$

rechtferigt dies. Dann folgt aber auch, daß

$$r_{k+1} - bt_{k+1} = -q_k r_k + r_{k-1} - b \cdot (t_{k-1} - q_k t_k)$$

durch a teilbar ist (der Induktionsschritt geht von $k-1 und k$ zu $k+1$).
Am Ende ist also $d = r_{k+1}$ und mit $v = t_{k+1}$

$$au = d - bv,$$

woraus u berechnet werden kann.

6.1.3 Ausrollen von Schleifen

Schauen Sie sich das folgende Programm an. In beiden Schleifen werden dieselben Rechenoperationen durchgeführt. In der zweiten Schleife werden mehrere nacheinanderfolgende Operationen zusammengefaßt und der Zählindex entsprechend vergrößert.

```
import java.io.*;
import java.lang.Math.*;

public class ausrollen
{
  public static int nn =  100 * 65536;   // durch 8 teilbar !
  public static byte a[] = new byte[nn];
  public static int s;

public static void init()
{
  for (int i = 0; i < nn; i++)
    a[i] = (byte)(Math.random() * 100);
}

public static int summe1()
{
  s = 0;
  for (int i = 0; i < nn; i++)
    s = s + a[i];
  return s;
}

public static int summe2()
{
  s = 0;
  for (int i = 0; i < nn - 7; i = i + 8)
    s = s+a[i]+a[i+1]+a[i+2]+a[i+3]+a[i+4]+a[i+5]+a[i+6]+a[i+7];
  return s;
}

public static void main(String arg[])
{
```

```
  long t = B.zeit(0);
  init();
  int s;
  t = B.zeit(t);
  s = summe1();
  t = B.zeit(t);
  System.out.println(s);
  s = summe2();
  t = B.zeit(t);
  System.out.println(s + "  ");
}
}
```

Es treten zwei Effekte auf:

1. Die zweite Schleife benötigt bei Fortran nur etwa die Hälfte der Rechenzeit; dies trifft bei sehr langen Schleifen auch bei Pascal zu. Bei Java ist der Beschleunigungseffekt auch zu messen, ich konnte ihn nur nicht so deutlich nachweisen. Das liegt daran, daß Felder nicht beliebig groß gemacht werden können: Die Indizierung hat durch `int`-Zahlen zu erfolgen; wenn man ein Feld anlegen will, für das der Speicher nicht ausreicht, erhält man nicht etwa bei der Übersetzung eine Fehlermitteilung, sondern beim Versuch, die Klasse abarbeiten zu lassen, „findet“ Java die Klasse nicht.

   Wenn ich wie oben ein Feld aus 10 Mio Bytes aufsummiere, braucht mein Rechner 3 bzw. 2 Sekunden. (Allerdings dauerte die Initialisierung über eine Minute.) Da `float`-Felder nicht so groß werden können, ergaben sich jeweils Rechenzeiten von 0 Sekunden.

   Wenn allerdings in der Schleife kompliziertere Operationen (Funktionsaufrufe) ausgeführt werden, geht die Beschleunigung zurück.

2. Die Ergebnisse der Summation unterschieden sich bei Fortran in der letzten Dezimalstelle. Welches Ergebnis war wohl richtiger als das andere?

   Hier sind alle Summanden positiv und innerhalb eines Achterblocks wahrscheinlich ungefähr von derselben Größenordnung, so daß die Zahl der abgeschnittenen Stellen gering bleiben könnte. Die Zahl s wird aber ständig größer, so daß sich s bei kleinen a_i evtl. gar nicht verändert. Wahrscheinlich ist die zweite Rechnung vorzuziehen. (Wenn die Summe der ersten 2^{16} natürlichen Zahlen im `real*4`-Bereich berechnet wird, so ist beim ersten Verfahren die fünfte (von 6 bis 7) Dezimalstellen falsch, beim zweiten richtig).

Wenn die Länge der auszurollenden Schleife nicht durch die Ausroll-Breite teilbar ist, muß das Schleifenende natürlich nachbehandelt werden.

6.1.4 Komplexe Zahlen

Eine komplexe Zahl hat die Gestalt $z = a + bi$, wobei a, b reell sind und $i^2 = -1$ gilt; diese Darstellung mit Real- und Imaginärteil heißt die kartesische Darstellung von z. Für die Polarkoordinaten des Punkts (a, b) gilt

$$a = r \cos \varphi$$

$$b = r \sin \varphi,$$

dabei ist $r = |z| = \sqrt{a^2 + b^2}$ und φ das Argument von z. Bei der Berechnung des Betrags kann ein Überlauf vorkommen, den man dadurch abfangen kann, daß man $\max(|a|, |b|)$ ausklammert. Das Argument bestimmt man am einfachsten durch $\varphi = \arccos(a/r)$.
Während sich die Addition und Subtraktion komplexer Zahlen am einfachsten in der kartesischen Darstellung durchführen läßt, sind die Multiplikation, Division, das Wurzelziehen oder allgemeiner die Potenzierung in der Polardarstellung einfacher: bei der Multiplikation multiplizieren sich die Beträge und addieren sich die Argumente.

6.2 Polynome

Werte berechnen

Für das Polynom $f(x) = a_0 x^n + a_1 x^{n-1} + \cdots a_{n-1} x + a_n$ soll der Wert $f(b)$ berechnet werden.
Die naive Methode geht so:

$w := a_0$
für $i = n - 1 :\ w := w + a_i * Math.pow(b, i)$

Aufwand: Falls der Compiler pfiffigerweise die Potenzberechnung in $b * b \cdots * b$ verwandelt, also keine Potenzreihenentwicklung durchführt, sind $1 + 2 + \cdots + n = \frac{n(n+1)}{2} \sim n^2$ Multiplikationen nötig.

Das Schema von Horner (1819) benötigt nur n Multiplikationen, denn wegen

$$f(b) = (\cdots((a_0 b + a_1)b + a_2)b + \cdots a_{n-1})b + a_n$$

kann man so rechnen:

$w := a_0$
für $i = 1, \ldots, n :$
$w := w * b + a_i$

Potenzieren

Dieselbe Idee kann man verwenden, um (große) Potenzen einer Zahl zu berechnen. Um x^n zu berechnen, stellen wir den Exponenten im Binärsystem dar: $n = (b_k \ldots b_0)_2 = \sum b_i 2^i$. Wir sehen also, daß n der Wert eines gewissen Polynoms an der Stelle 2 ist. Ein Summand (1) verlangt eine Multiplikation, ein Faktor (2) verlangt eine Quadrierung.

```
  y = 1
  w = x
  solange n > 0:
    wenn n ungerade:
      y = y * w
    wenn n > 1:
      w = w * w
      n = n / 2
Das Ergebnis ist y
```

Dies entspricht der Auswertung des Exponenten mittels des Horner-Schemas: Wir stellung uns die binär dargestellte Zahl als Polynom vor, dessen Koeffizienten Nullen und Einsen sind, in das die Zahl 2 eingesetzt ist.

Polynomdivision mit Rest

Seien $f(x)$, $g(x)$ Polynome, dann gibt es eindeutig bestimmte Polynome $q(x)$, $r(x)$ mit

1. $f(x) = g(x)q(x) + r(x)$ und
2. $r(x) = 0$ oder $\deg(r) < \deg(g)$.

Das Polynom $q(x)$ heißt der Quotient, $r(x)$ der Rest der Polynomdivision.

Falls nun $g(x) = x - b$ ein lineares Polynom ist, dann ist der Divisionsrest eine Konstante, und zwar gleich $f(b)$, wie man aus

$$f(x) = (x - b)q(x) + r$$

durch Einsetzen von $x = b$ sieht.

Das wäre eine neue Methode der Wertberechnung. Wie effektiv ist sie?
Wir werden gleich sehen, daß man mittels einer kleinen Modifizierung des Hornerschemas den Quotienten und den Rest der Polynomdivision berechnen kann.

Sei wieder $f(x) = a_0x^n + a_1x^{n-1} + \cdots a_{n-1}x + a_n$. In unserem Fall hat der Quotient den Grad $n - 1$; wir machen den Ansatz

$$q(x) = b_0x^{n-1} + \cdots + b_{n-k-1}x^k + \cdots + b_kx^{n-k-1} + \cdots + b_{n-1}$$

und vergleichen in

$$f(x) = q(x)(x - b) + b_n$$

die Koeffizenten der x-Potenzen (den Rest nennen wir b_n; gelegentlich ist eine geschickte Notation hilfreich):

$$a_0 = b_0$$

$$x^{n-k}: \qquad a_k = b_k - b_{k-1}b$$

Damit erhalten wir schrittweise die Koeffizienten von $q(x)$:

$$b_0 = a_0$$

$$b_k = a_k + b_{k-1}b,\ k = 1, \ldots, n$$

und $b_n = f(b)$ ist der Divisionsrest. Der Rechenaufwand ist derselbe wie beim Hornerschema.

Wenn man eine reelle Nullstelle b des Polynoms $f(x)$ bestimmt hat, so kann man dieses Verfahren nutzen, um den Faktor $x-b$ aus $f(x)$ zu entfernen und kann nach Nullstellen des Quotienten suchen.

Wenn $a+bi$ eine komplexe Nullstelle von $f(x)$ ist, so ist auch $a-bi$ eine Nullstelle von $f(x)$, d.h. $f(x)$ ist durch das quadratische (reelle) Polynom $(x-a-bi)(x-a+bi) = x^2-2ax+a^2+b^2$ teilbar. Um diesen Faktor aus $f(x)$ zu entfernen, verwenden wir eine Verallgemeinerung des Hornerschemas (Collatz 1940):

Wir berechnen also den Quotienten und den Rest bei der Division von $f(x)$ durch x^2+px+t. Der Quotient sei

$$q(x) = b_o x^{n-2} + b_1 x^{n-3} + \cdots + b_{n-2},$$

der Rest sei

$$r(x) = b_{n-1}x + b_n.$$

Der Koeffizientenvergleich in $f(x) = (x^2+px+t)q(x) + r(x)$ liefert

$$\begin{aligned}
a_0 &= b_0 \\
a_1 &= b_1 + pb_0 \\
&\cdots \\
a_k &= b_k + pb_{k-1} + tb_{k-2}, \qquad k = 2, \ldots, n-1 \\
&\cdots \\
a_n &= tb_{n-2} + b_n
\end{aligned}$$

also

$$\begin{aligned}
b_0 &= a_0 \\
b_1 &= a_1 - pb_0 \\
&\cdots \\
b_k &= a_k - pb_{k-1} - tb_{k-2}, \qquad k = 2, \ldots, n-1 \\
&\cdots \\
b_n &= a_n - tb_{n-2}
\end{aligned}$$

Um die vorgestellten Verfahren zur Nullstellenabtrennung nutzen zu können, muß man zuerst eine Nullstelle kennen. Im Buch von *Beresin und Shidkov, Numerische Methoden 2, VEB Deutscher Verlag der Wissenschaften, Berlin 1971* wird auf S. 169 ein Verfahren von Muller (1956) vorgestellt, mit dessen Hilfe komplexe Nullstellen von Polynomem berechnet werden können. Der Witz besteht darin, daß (anders als beim Newton-Verfahren) keine Anfangsnäherung bekannt sein muß; das Verfahren ist zwar theoretisch nicht streng abgesichert, liefert aber in der Praxis gute Ergebnisse.

Für das Verfahren von Graeffe zur Bestimmung reeller Nullstellen habe ich z.Z. nur die Referenz auf *Zurmühl, Praktische Mathematik für Ingernieure und Physiker, Springer 1957, S. 62).*Zur Bestimmung komplexer Nullstellen gibt es Verfahren von Nickel bzw. Laguerre, die z.B. bei *Gander, Computermathematik, Birkhäuser 1992, S. 110 ff* vorgestellt werden.

Verwenden Sie bitte diese Algorithmen, um Nullstellen zu berechnen und abzutrennen. Berechnen Sie z.B. die Nullstellen von $x^n - 1$, dies sind (für geeignetes n) die Positionen der Sterne im EU-Wappen.

6.3 Auswertung vieler Polynome

Wenn nicht der Wert eines Polynoms an einer Stelle, sondern die Werte vieler Polynome vom Grad $\leq n$ an denselben Stellen $x_0, x_1, \ldots, x_n$ berechnet werden sollen, so ist es besser, sich zuerst die Potenzen x_i^j bereitzustellen. Für das Polynom $p(x) = \sum p_i x^i$ gilt dann:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ & \ldots & & & \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_n \end{pmatrix} = \begin{pmatrix} p(x_0) \\ p(x_1) \\ \vdots \\ p(x_n) \end{pmatrix}.$$

Umgekehrt kann man bei vorgegebenen Werten $p(x_i)$ die Koeffizienten des Polynoms p durch Multiplikation mit der zu (x_i^j) inversen Matrix bestimmen.
Das vereinfacht das formale Rechnen mit Polynomen kollossal: Um das Produkt (die Summe) von $p(x)$ und $q(x)$ zu bestimmen, berechnet man einfach $p(x_i) \cdot q(x_i)$ und bestimmt die Koeffizienten des Polynoms mit diesen Werten. Das lohnt sich natürlich erst dann, wenn viele Opeartionen mit Polynomen durchgeführt werden: Am Anfang wird jedes Polynom durch seinen „Wertevektor“ kodiert, am Ende werden die Koeffizientenvektoren ermittelt.
Wenn für die x_i die $(n+1)$-sten Einheitswurzeln gewählt werden, so ist die Matrix (ω^{ij}) zu betrachten; deren Inverse ist ganz einfach aufgebaut: $\frac{1}{n+1}(\omega^{-ij})$. Dieses Verfahren heißt „Diskrete Fourier-Transformation“ (DFT).

6.4 Matrixoperationen

Wir wollen uns zunächst überlegen, was zu tun ist, wenn auf die Komponenten eines zweidimensionalen Feldes zugegriffen werden soll. Dabei soll `xxx` ein Datentyp sein, der `w` Bytes belegt.

```
character[][] = new  a[m][n];
```

Die Zahl der Spalten der Matrix ist n, die Anfangsadresse sei b; dann hat die Komponente a_{ij} (bei zeilenweiser Abspeicherung) die Adresse

$$s = b + (j \cdot n + i) \cdot w,$$

zur Laufzeit werden also 4 Rechenoperationen (für den Zugriff!) durchgeführt.

Diese Überlegungen könnten interessant sein, wenn man häufig auf Teilfelder zugreifen will und sich ein mehrdimensionales Feld selbst als eindimensionales organisiert.

Ein Beispiel: Matrixmultiplikation

(Es sei `a` eine (m, n)-Matrix und `b` eine (n, q)-Matrix. Die (m, q)-Matrix `c` wird jeweils vorher als Nullmatrix initialisiert.)

```
for (k = ...)                  // in der inneren Schleife
  for (j = ...)                // wird viermal dereferenziert
    for (i = ...)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];

for (k = ...)
  for (j= ...)
  {
    s = b[k][j];               // dadurch wird etwa 1/6 der
    for (i = ...)              // Rechenzeit gespart
      c[i][j]= c[i][j] + a[i][k] * s;
  }

for (i = ...)
  for (j = ...)
  {
    s = 0.0;                   // dadurch wird etwa 1/3 der }
    for (k = ...)              // Rechenzeit gespart }
      s = s + a[i][k] * b[k][j];
    c[i][j] = s;
  }
```

Die gesparte Rechenzeit ist eben die zum Dereferenzieren benötigte. Die letzte Möglichkeit ist aber nur bei der dort gegebenen Reihenfolge der Schleifen möglich. Ein guter Compiler optimiert alledings den Rechenablauf neu und ganz anders, als man es gedacht hat.

Wir wollen nun untersuchen, wie viele Additionen/Subtraktionen und Multiplikationen/Divisionen zur Lösung eines linearen Gleichungssystems nötig sind. Gegeben sei ein Gleichungssystem $Ax = b$, wobei A eine invertierbare $n \times n$-Matrix ist; das Gleichungssystem hat also eine eindeutig bestimmte Lösung. Wir verwenden als Lösungsverfahren die Gauß-Jordan-Elimination und setzen der Einfachheit halber voraus, daß dabei keine Zeilenvertauschungen durchgeführt werden müssen.
Durch Addition von Vielfachen einer Zeile zu anderen wird die erweiterte Koeffizientenmatrix $(A \mid b)$ zunächst in eine Dreiecksform

$$\begin{pmatrix} 1 & a_{12} & \dots & a_{1n} & b_1 \\ 0 & 1 & a_{23} & \dots & b_2 \\ & \dots & & & \\ 0 & \dots & 0 & 1 & b_n \end{pmatrix}$$

und danach durch Rücksubstitution in die Form

$$\begin{pmatrix} 1 & 0 & \dots & 0 & b_1 \\ 0 & 1 & \dots & 0 & b_2 \\ & \dots & & & \\ 0 & \dots & 0 & 1 & b_n \end{pmatrix},$$

gebracht, dann ist das Lösungstupel sofort ablesbar.

Die folgenden elementaren Überlegungen sind im Lehrbuch von H. Anton, Lineare Algebra, Spektrum 1995 auf den Seiten 538 ff. ausführlich erläutert.

Schritt	Mult	Add	Bemerkung
1	n	0	Erzeugen der Anfangseins
	$n(n-1)$	$n(n-1)$	Nullen in $n-1$ Zeilen unter der 1
2	$n-1$	0	Erzeugen der Anfangseins
	$(n-1)(n-2)$	$(n-1)(n-2)$	Nullen in $n-2$ Zeilen unter der 1
			...
$n-1$	2	0	Erzeugen der Anfangseins
	2	2	Null in Zeile unter der 1
			Rückwärtsphase
1	$n-1$	$n-1$	Nullen in $n-1$ Zeilen über der 1
2	$n-2$	$n-2$	Nullen in $n-2$ Zeilen über der 1
			...
$n-1$	1	1	Null in Zeile über der 1

Somit erhalten wir als Anzahl der Additionen

$$\sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i + \sum_{i=1}^{n-1} i = n^3/3 + n^2/2 - 5n/6$$

und als Anzahl der Multiplikationen

$$\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n-1} i = n^3/3 + n^2 - n/3,$$

insgesamt als ca. n^3 Rechenoperationen.

Bemerkung: Wenn sofort auch über den Einsen Nullen erzeugt werden, sind $\frac{1}{2}(n^3 - n)$ Additionen und $\frac{1}{2}(n^3 + n)$ Multiplikationen nötig.

Schließlich geben wir einige Typen von Matrizen an, die leicht zu erzeugen sind und wo für die Determinante bzw. die Inverse explizite Formeln bekannt sind. Damit kann man eigene Algorithmen testen.

Typ		Determinante	Inverse
Vandermonde	(x_i^j)	$\prod x_i \cdot \prod_{i<j}(x_j - x_i)$	
kombinatorische	$(y + \delta_{ij} \cdot x)$	$x^{n-1} \cdot (x + ny)$	$\left(\frac{-y + \delta_{ij} \cdot (x + ny)}{x(x + ny)}\right)$
Cauchy	$\left(\frac{1}{x_i + y_j}\right)$	$\frac{\prod_{i<j}(x_j - x_i)(y_j - y_i)}{\prod_{i,j}(x_i + y_j)}$	$\left(\frac{\prod_k (x_j + y_k)(x_k + y_i)}{(x_j + y_i)\prod_{k\neq j}(x_j - x_k)\prod_{k\neq i}(y_i - y_k)}\right)$
Hilbert	$\left(\frac{1}{i + j - 1}\right)$		Spezialfall von Cauchy

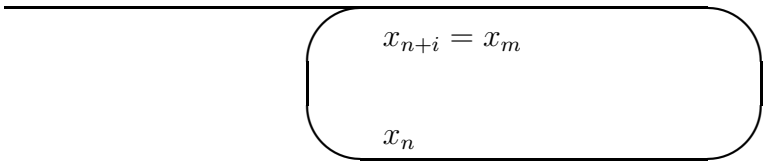
Die Inverse einer Hilbert-Matrix hat ganzzahlige Einträge; Hilbert-Matrizen werden gern für Tests genutzt, denn dieser Typ ist *numerisch instabil*.

6.5 Zufallszahlen

Ein beliebtes Mittel, um einen Algorithmus auf Korrektheit zu „testen“, ist es, ein Zahlenbeispiel mit der Hand durchzurechnen und nachzusehen, ob das Programm dasselbe Ergebnis liefert. Man kann auch viele Eingaben erzeugen und abwarten, ob der Rechner abstürzt oder nicht. Wenn das aber bei einigen Eingaben korrekt abläuft und bei anderen nicht, dann hat man ein Problem.
Es ist mitunter hilfreich, wenn man eine „zufällige“ Folge von Zahlen erzeugen kann, deren Erzeugung aber wiederholbar ist.
Ein klassisches Verfahren stammt von J. v. Neuman (1946): der „middle square“ - Algorithmus. Wir rechnen mit 4-stelligen Zahlen, beginnen mit einer Zahl, quadrieren und entnehmen die mittleren vier Dezimalstellen:

$$x = x^2/10^2 \bmod 10^4.$$

Es kann allerdings passieren, daß man in kurze Zyklen gerät, z.B. ist 6100 eine wenig geignete Startzahl (6100, 2100, 4100, 8100, 6100). Wie kann man die Güte eines Zufallszahlen-Generators überprüfen?
Sei M eine endliche Menge, $f : M \longrightarrow M$ eine Abbildung und $x_0 \in M$ ein Startwert. Man bildet nun $x_{i+1} = f(x_i)$. Wegen der Endlichkeit von M können nicht alle x_i voneinander verschieden sein. Es sieht etwa so aus:



Es gibt also eine Periode, die bei einem Wert $x_m, m \geq 0$ beginnt und die Länge $l \geq 1$ hat. Es gilt also

$$x_i = x_{i+l} \text{ für } i \geq m.$$

Der durch f gegebene Algorithmus ist „gut“, wenn desses Periodenlänge groß ist, die Länge der Vorperiode ist unerheblich. Wie können diese Daten bestimmt werden?

Wir bemerken zuerst, daß es eine Zahl n gibt, für die $x_n = x_{2n}$ gilt (solch ein x heißt idempotent).
Denn es ist $x_r = x_n$ gdw. $r - n = k \cdot l$ ein Vielfaches von l und $r > n \geq m$ ist. Wir suchen also das kleinste n mit $n = kl$, dann ist $x_{2n} = x_{n+kl} = x_n$.

Aus diesen Idempotenz-Index n können nun die Werte von m und l bestimmt werden: Wenn wir die Vorperiode auf die Periode „aufwickeln“, so liegt deren Startpunkt gerade bei x_n. Also suchen wir das kleinste i mit $x_i = x_{n+i}$, dies ist unser m.
Wenn die Vorperiode m nicht um die Periode herumreicht ($k = 1$), so ist die Periodenlänge $l = n$, andernfalls ist l gleich dem kleinsten i mit $x_{m+i} = x_m$.

Ein einfacher Generator ist die Funktion

$$f(x) = x \cdot 5 \bmod 97, \text{ also } x_n \equiv 5^n,$$

er hat keine Vorperiode und die (maximal mögliche) Länge 96 (warum ?).

In Fortran sah das so aus: (Wir suchen nacheinander i mit $f^i(x) = f^{2i}(x)$, dann ist $n = i$; $f^i(x) = f^{n+i}(x)$; dann ist $m = i$; $f^m(x) = f^{m+i}(x)$; dann ist $l = i$.

```
      program rand
c     Periode eines Zufallsgenerators: Vorperiode: m; Periode: l
      integer m, f2n, s, h, x, i, n, l
      parameter (h = 97)

      do s = 2, h
        print *, "s = ", s
        do i = 1, h
          if (f2n(x, s, h, i) .eq. f2n(x,s,h, 2*i)) then
            print *,  "Idempotent", i, f2n(x,s,h,i), f2n(x,s,h,2*i)
            n = i
            goto 1
          end if
        end do

1       do i = 1, h
          if (f2n(x, s, h, i) .eq. f2n(x, s, h, n+i)) then
            m = i
            goto 2
          end if
        end do

2       print *, "Vorperiode ", m
        print *, "x^m = ", f2n(x, s, h, m)
        do i = 1, m
          if (f2n(x,s,h,i) .eq. f2n(x,s,h,n+i)) then
            l = i
```

```
          goto 3
        else
          l = n
        end if
       end do

3      print *, "Laenge ", l
       print *, "Kontrolle : "
       print *,(i,":", f2n(x,s,h,i), i=1,50)
       read *
       end do
      end

      integer function f(x, s, h)
      integer x, s, h
        f = mod(x*s, h)
      end

      integer function f2n(x, s, h, n)
c   f2n(x) = f^n(x)
      integer x, s, h, n, i, r, f, y
        y = x
        do i = 1, n
          r = f(y, s, h)
          y = r
        end do
        f2n = y
      end
```

7 Suchen

Komplexität
Sortieralgorithmen reagieren unterschiedlich, wenn die zu sortierende Menge mehr oder weniger vorsortiert ist. Es kommt also bei der Bewertung von Algorithmen nicht auf die Laufzeit bei einer konkreten Eingabe, sondern auf die maximale oder auch die mittlere Laufzeit bei beliebigen Eingaben der Größe n an. Die Laufzeit eines Algorithmus A ist also eine Funktion $L_A : N \longrightarrow N$, für die man eine obere bzw. untere Schranke bestimmen möchte.
Definition:
Eine Funktion $f : N \longrightarrow N$ ist eine obere Schranke für die Funktionenmenge

$$O(f) = \{g : N \longrightarrow N \mid \text{ es gibt } c > 0 \text{ und } n_0 \text{ mit } g(n) \leq c \cdot f(n) \text{ für alle } n > n_0\}.$$

Eine Funktion $f : N \longrightarrow N$ ist eine untere Schranke für die Funktionenmenge

$$U(f) = \{g : N \longrightarrow N \mid \text{ es gibt } c > 0 \text{ und } n_0 \text{ mit } g(n) \geq c \cdot f(n) \text{ für alle } n > n_0\}.$$

Eine Funktion $f : N \longrightarrow N$ ist eine exakte Schranke für die Funktionenmenge

$$E(f) = \{g : N \longrightarrow N \mid \text{ es gibt } c > 0 \text{ und } n_0 \text{ mit } \frac{1}{c} \cdot f(n) \leq g(n) \leq c \cdot f(n) \text{ für alle } n > n_0\}.$$

Ein Algorithmus A heißt linear zeitbeschränkt, wenn $L_A \in O(n)$ ist, er heißt polynomial zeitbeschränkt, wenn ein $k \in N$ existiert, so daß $L_A \in O(n^k)$ ist, und er heißt exponentiell zeitbeschränkt, wenn ein k existiert, so daß $L_A \in O(k^n)$ gilt.
Wir werden uns bei konkreten Algorithmen auch für ihre Zeitkomplexität interessieren. In der Praxis sind bei großen Datenmengen nur Algorithmen einsetzbar, die höchstens polynomiale Zeitschranken haben, und dabei muß der Exponent noch „klein“ sein. Bedenken Sie: $n = 3.5 \cdot 10^6$ (die Zahl der Transistoren auf einem modernen Mikrochip) und $k = 4$ ergibt $n^k = 1.5 \cdot 10^{26}$, aber 10^{17} Nanosekunden sind 3 Jahre.

Die Entwicklung der letzten Jahrzehnte verlief so, daß alle 10 Jahre 100mal mehr Speicher zur Verfügung stand und die Rechengeschwindigkeit verhundertfacht wurde. Welche Auswirkung hat das auf die Größe der bearbeitbaren Probleme?

Ein Algorithmus habe polynomiale Komplexität: $O(n^p) \sim C \cdot n^p$. Wenn ein neuer Computer eingesetzt wird, der um den Faktor k besser ist, so kann man Probleme der Größe kn bearbeiten. Dazu sind dann $C(kn)^p$ Operationen nötig, die Rechenzeit ist dann also

$$C(kn)^p/k = Ck^{p-1}n^p.$$

Wenn also p größer als 1 ist, hat man Probleme.

Als Beipiele betrachten wir einige Suchalgorithmen.

Lineares Suchen

Wenn ein Wert W in einer Liste L der Länge n, über deren Inhalt nichts weiter bekannt ist, gesucht werden soll, so muß man die Liste sequentiell durchlaufen:

```
i:= 1
solange i <= n und L[i] <> W ist, erhoehe i.
Wenn i <= n ist, so wurde W gefunden.
```

Aufwand: Wenn W nicht in der Liste vorkommt, sind n Schritte nötig, wenn W vorkommt, sind maximal $n - 1$ Schritte notwendig, im Durchschnitt werden es $\frac{n-1}{2}$ sein. Der Zeitaufwand ist linear.

Man kann das Zeitverhalten noch verbessern:

```
i:= 1
L[n+1]:= W
solange L[i] <> W ist, erhoehe i.
Wenn i <= n ist, so wurde W gefunden.
```

Man hat den Test, ob die Feldgrenze erreicht ist, eingespart. Dieser Test ist nötig, wenn man nicht weiß, ob W in der Liste vorkommt.

Binäres Suchen

In geordneten Strukturen geht das Suchen schneller.
Beispiel: Ich denke mir eine natürliche Zahl zwischen 0 und 100. Wenn man fragt: „Ist es 1, ist es 2, . . . ?“, so hat man sie nach einiger Zeit auf alle Fälle gefunden. Wenn man fragt: „Ist es kleiner, gleich oder größer als 50?“, so hat man in Abhängigkeit von der Antwort die zu durchsuchende Menge in einem Schritt halbiert. Eine Zahl unter 100 errät man also in höchstens 7 Schritten.
Dieses Verfahren heißt binäres Suchen, es ist ein Beipiel für das Prinzip des Teilens und Herrschens. Wir implementieren den Algorithmus zuerst rekursiv`such1` und betrachten den Zeitaufwand: Bei jedem neuen Aufruf von `such1` halbiert sich der Aufwand. Wenn $2^{m-1} < n \leq 2^m$, so sind $m = \log_2 n$, Schritte nötig. Mit gleichem Zeitaufwand arbeitet die sequentielle Prozedur `such2`; wir beschleunigen sie durch den Trick, in jeder Schleife nur eine Abfrage durchzuführen:

```
import java.io.*;
import java.lang.Math.*;

public class suchen
{
  public static int nn =  92;
  public static long a[] = new long[nn];
  public static long s;
// Die Position des Werts s soll im geordnetet Feld a gefunden werden.

public static void init()
{
  a[0] = 1; a[1] = 1;
  for (int i = 2; i < nn; i++)
    a[i] = a[i-2] + a[i-1];
}

public static int such1(int links, int rechts)
{
  int i;
  System.out.print("+");
  if (links <= rechts)
  {
    i = (links + rechts) / 2;
    if (a[i] > s)
      return such1(links, i-1);
    else if (a[i] < s)
      return such1(i+1, rechts);
    else
      return i;   // gefunden !
  }
  else
    return -1;    // nicht gefunden !
```

```
}

public static int such2()
{
  int links = 0, rechts = nn-1, i;
  while (links < rechts)
  {
    System.out.print("*");
    i = (links + rechts) / 2;
    if (a[i] < s)
      links = i + 1;
    else
      rechts = i;
  }
  if (a[rechts] == s)
    return rechts;
  else
    return -2;
}

public static void main(String arg[])
{
  init();
  s = 13;
  int i;
  i = such1(0,nn-1);
  System.out.println(i);
  i = such2();
  System.out.println(i);
}
}
```

Hashing

Das Suchen in binären Bäumen entspricht dem Auffinden der richtigen Antwort durch ja/nein-Entscheidungen. Komplexere Entscheidungen können aber evtl. schneller zum Ziel führen: Die Telefonnummer unserer Sekretärin, Frau Zyska, würde ich nicht suchen, indem ich das Telfonbuch zuerst in der Mitte aufschlage, sondern am Ende.
Also: Wir legen für jeden Eintrag einen Schlüssel fest, der die ungefähre Position in der Tabelle festlegt.
In Suchbäumen oder Listen ist die Zuordnung: Eintragung $\mapsto$ Standort injektiv. Wir betrachten nun eine Hash-Funktion $h : \{\text{Einträge}\} \longrightarrow \{\text{Indizes}\}$, die nicht notwendigerweise injektiv sein muß, für die aber die Urbildmengen $h^{-1}(i)$ für alle i ungefähr gleich groß sein sollen.
Wenn die zu verwaltenden Daten (kleine) Zahlen sind, so kann man sie selbst als Index in einem Feld verwenden und der Zugriff ist trivial. Wenn die Einträge aber Zeichen-

ketten aus den Buchstaben a, ..., z sind und die Wortlänge auf 16 festgelegt ist, so gibt es $26^{16} = 4 \cdot 10^{22}$ Möglichkeiten, es gibt aber nur 10^5 vernünftige Worte, man hätte also einen 10^{17}-fachen Aufwand betrieben. (Man veranschauliche sich die Größenordnung: das Weltall existiert seit 10^{16} Sekunden; dabei sind die längeren Schaltjahre noch gar nicht berücksichtigt.)

Bei Kollisionen ($h(s) = h(t), s \neq t$) reicht es nicht aus, in einer Tabelle die zu s gehörige Information abzulegen, man braucht auch s selbst, um eine aufgetretenen Kollision erkennen zu können. Außerdem muß man eine Strategie zur Kollisionsauflösung entwickeln.

Es folgen einige Beispiele für Hash-Funktionen zur Verwaltung von Zeichenketten $Z = (z_1, \ldots, z_k)$ mit einem Indexbereich $I = [0 \ldots n]$:

1. $h(z) = \sum_{i=1}^{k} (p_i \cdot (int)(z_i) \bmod n)$, wo p_i die i-te Primzahl und $(int)(A)$ der ASCII-Index ist. Diese Funktion erzeugt wenig Kollisionen, ist aber sehr rechenaufwendig.

2. $h(z) = (int)(z_1)$; Ersetze $h := (256 \cdot h + (int)(z_i)) \bmod n$ für $i = 2$ bis k. Dabei sollte $\mathrm{ggT}(n, 256) = 1$ sein. Die Multiplikation mit 256 ist einfach eine Verschiebung um 8 Bit und dürfte schnell sein.

3. Wir verwenden einzelne Zeichen und die Länge von z:
   $h(z) = (7 \cdot (int)(z_1) + 11 \cdot (int)(z_{k-1}) + 19k) \bmod n$.

Die Länge n der Tabelle sollte eine Primzahl sein. Die Felder $a[1], \ldots, a[n]$ können dann Listen enthalten, die die zu speichernden Informationen enthalten, innerhalb dieser (hoffentlich kurzen) Listen kann man dann suchen.

Teilen und Herrschen

Dies ist eine oftmals effektive Lösungsmethode.
Sei ein Problem der Größe n zu lösen. Wir suchen eine obere Schranke $L(n)$ für die benötigte Laufzeit. Wir nehmen an, das Problem lasse sich in t Teilprobleme desselben Typs zerlegen, diese Teilprobleme mögen die Größe $\frac{n}{s}$ haben und zum Zerlegen und Zusammenfügen sei der Zeitaufwand $c(n)$ nötig. Wir haben also folgende Rekursion zu bearbeiten:

$$L(n) = \begin{cases} c(n) & n = 1 \\ t \cdot L(\frac{n}{s}) + c(n) & \text{sonst} \end{cases}$$

Satz 7.1 *Für $n = s^k$ gilt $L(n) = \sum_{i=0}^{k} t^i c(\frac{n}{s^i})$.*

Beweis: Wir führen die Induktion über k. Für $k = 0$, also $n = 1$ ist $L(1) = c(n)$ gültig. Weiter gilt:

$$L(n) = L(s^k) = t \cdot L(\frac{n}{s}) + c(n) = t \sum_{i=0}^{k-1} t^i c(\frac{\frac{n}{s}}{s^i}) + c(n) = \sum_{i=0}^{k-1} t^{i+1} c(\frac{n}{s^{i+1}}) + c(n) = \sum_{i=0}^{k} t^i c(\frac{n}{s^i}).$$

□

Satz 7.2 *Wenn $c(n) = c \cdot n$ gilt, so ist*

$$L(n) = \begin{cases} O(n) & t < s \\ O(n \log n) & t = s \\ O(n^{\log_s t}) & t > s \end{cases}$$

Beweis: Es gilt also

$$L(n) = \sum_{1=0}^{k} t^i c \frac{n}{s^i} = nc \sum_{i=0}^{k} (\frac{t}{s})^i.$$

Für $t < s$ ist die Summe durch $K = \frac{1}{1 - \frac{t}{s}}$ beschränkt (geometrische Reihe), also ist $L(n) \leq c \cdot K \cdot n = O(n)$. Für $t = s$ gilt $L(n) = cn(k+1) = O(n \log n)$. Für $t > s$ gilt

$$L(n) = cn \sum_{i=0}^{n} (\frac{t}{s})^i = cn \frac{(\frac{t}{s})^{k+1} - 1}{\frac{t}{s} - 1} = O(s^k (\frac{t}{s})^k) = O(t^{\log_s n}) = O(n^{\log_s t}).$$

Die letzte Identität erkennt man, indem man auf beiden Seiten $\log_s$ bildet. □

8 Einfache Datenstrukturen und ihre Implementation

Die einfachste Datentruktur ist eine lineare Liste. Wir haben $n > 0$ Einträge x[1], x[2], ..., x[n]. Über die Struktur der gespeicherten Daten machen wir uns keine Gedanken. Die Grundaufgaben zur Bearbeitung solchen Listen sind die folgenden:

- Zugriff auf den k-ten Eintrag, Auswertung, Änderung.
- Vor dem k-ten Eintrag ist ein neuer einzufügen.
- Löschen des k-ten Eintrags.
- Zwei Listen sollen zu einer vereinigt werden.
- Eine Liste ist in zwei Teillisten zu zerlegen.
- Eine Kopie einer Liste soll erstellt werden.
- Die Länge einer Liste ist zu bestimmen.
- Die Einträge einer Liste sollen geordnet werden.
- Eine Liste soll nach speziellen Einträgen durchsucht werden.

In Abhängikeit davon, welche Operationen im Einzelfall mehr oder weniger häufig auszuführen sind, gibt es unterschiedliche Darstellungen von Listen in Rechnern, die für eine Operation mehr oder weniger effektiv sind.
Für Listen, wo der Zugriff (Einfügen, Streichen) immer am Anfang oder am Ende der Liste stattfindet, gibt es spezielle Namen:

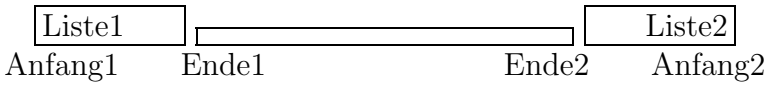
- Stack (Stapel, Keller): Einfügen und Streichen stets am Ende (filo),
- Queue (Warteschlange): Alle Einfügungen an einer Seite, alle Streichungen an der anderen Seite (fifo),
- Deque (double-ended queue): alle Zugriffe an den Enden.

Typische Anwendungen eines Stacks sind die folgenden: Eine Menge wird durchforstet, evtl. relevante Daten werden einstweilen beiseite (in den Keller) gelegt, um sie später zu verarbeiten. Beim Aufruf eines Unterprogramms werden die Registerinhalte auf einem Stack gerettet, das Unterprogramm kann die Register selbst nutzen, nach dem Verlassen des Unterprogramms werden die Daten aus dem Stack zurückgeholt.
Oder nehmen wir ein Abstellgleis. Es kommen vier Wagen in der Reihenfolge 1, 2, 3, 4 an und sollen in der Reihenfolge 2, 4, 3, 1 abfahren. Wagen 1 fährt aufs Abstellgleis, Wagen 2 auch, fährt aber gleich wieder runter. Nun fahren 3 und 4 aufs Abstellgleis, 4 fährt wieder runter, dann verlassen zuerst 3 und dann 1 das Gleis.
Kann man 123456 in 325641 überführen? Und in 154623?
Es kann nie die Situation $i < j < k$ und $p_k < p_i < p_j$ eintreten.

Wenn die Einträge in einer Liste alle die gleiche Anzahl C von Bytes belegen, kann man die Liste sequentiell abspeichern, dann gilt:

```
Addr(x[i+1]) = Addr(x[i]) + C
```

Wenn X einen Stack realisieren soll, so braucht man sich nur den aktuellen Index des Endes (den sog. stack pointer) zu merken, beim Einfügen ist er zu erhöhen, beim Streichen zu senken.
Bei einer Warteschlange müssen wir uns zwei Indizes A, E für Anfang und Ende merken (besser: den ersten freien Platz nach dem Ende). Wenn A = E gilt, so ist die Schlange leer. Wenn stets am Anfang gestrichen und am Ende eingefügt wird, so bewegt sich die Schlange in immerhöhere Speicherbereiche, während die unteren ungenutzt bleiben. Es ist dann besser, die Warteschlange als Ring zu organisieren. Wenn nun nach dem Streichen A = E gilt, so ist die Schlange leer. Wenn aber nach dem Einfügen A = E ist, so ist die Schlange voll.
Wenn man in einer leeren Liste streichen will, dann hat man einen Fehler gemacht. Wenn man in eine volle Liste noch etwas einfügen will, dann ist wahrscheinlich die zu bearbeitende Datenmenge zu groß geworden. Solche Überläufe muß man abfangen.
Eine andere Möglichkeit besteht darin, daß sich zwei Stacks den Speicher teilen:



Ein Überlauf tritt hier erst ein, wenn der Platz für beide Listen nicht mehr ausreicht. Wenn mehrere Stacks zu verwalten sind, dann geht das nicht so. Dennoch kann es nützlich sein, den vorhandenen Speicher gemeinsam zu nutzen: der i-te Stack liegt zwischen A[i] und E[i], wenn zwei Stacks zusammenstoßen, verschiebt man die Stacks. Dies wird in Knuth's Buch (Band 1) auf den Seiten 245 und 246 beschrieben, wir werden uns diesem Problem im Praktikum widmen.

Einige häufige Anwendung eines Stacks ist die Auflösung rekursiver Funktionsaufrufe. Man kann z.B. $n!$ folgendermaßen berechnen:

```
public static int fak(int n)
{
  if (n == 1)
    return 1;
  else
    return n * fak(n - 1);
}
```

Als allgemeines Schema zur Auflösung der rekursiven Aufrufe kann folgendes gelten: Man richtet sich einen Stapel (stack) ein, in dem bei jeden Prozeduraufruf Daten eingetragen und beim Verlassen der Prozedur gestrichen werden. Jedem Prozeduraufruf wird ein Fach (frame) auf dem Stapel zugeordnet, der die folgenden Informationen enthält:

head	Übergabeparameter von anderen Prozeduren
head - 1	aktuelle Parameter
head - 2	lokale Variable
head - 3	nach außen zu übergebende Parameter (Ergebnisse)
head - 4	Rücksprungadresse

Diese Fächer sind gleichgroß und liegen aufeinanderfolgend auf dem Stapel; die Verwaltung geschieht einfach durch einen Zeiger (stack pointer) auf die Stapelspitze. Jedes Fach enthält die Informationen für einen Prozeduraufruf.

Sei A die aktuell arbeitende Prozedur, dann sieht der Stapel so aus:

Fach fürs Hauptprogramm
⋮
Fach für die Prozedur, die A gerufen hat
Fach für A

Wenn nun A die Prozedur B (oder sich selbst mit neuen Parametern) aufruft, dann machen wir folgendes:

1. Ein neues Fach wird auf die Spitze des Stapels gelegt. Dort legt man (in einer Reihenfolge, die B bekannt ist) folgendes ab:
   (a) die Adressen der aktuellen Parameter von B (wenn ein Parameter ein Ausdruck ist, wird er zuerst ausgewertet und die Adresse des Ergebnisses übergeben; bei Feldern genügt die Anfangsadresse),
   (b) leerer Raum für die lokalen Variablen von B,
   (c) die Nummer der Anweisung, die A nach dem Aufruf von B durchzuführen hat (Rücksprungadresse).
2. Nun werden die Anweisungen von B ausgeführt. Die Adressen der Werte aller Parameter von B findet B durch Abzählen von der Stapelspitze aus (rückwärts).

3. Wenn B beendet ist, wird wie folgt die Kontrolle an A übergeben:
   (a) Die Rücksprungadresse steht im B-Fach.
   (b) Ggf. steht ein Funktionswert an der A bekannten Stelle.
   (c) Das B-Fach wird entfernt, indem einfach der Zeiger auf die Stapelspitze um die passende Größe zurückgesetzt wird.
   (d) Die Arbeit von A wird an der richtigen Stelle fortgesetzt.

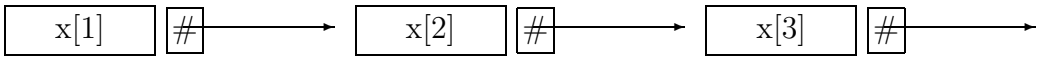
Für die folgende Fakultätsberechnung benötigen wir nicht alle diese Informationen.

```
import java.io.*;
import java.lang.Math.*;

public class fak
{
  public static int nn =  100;
  public static int a[] = new int[nn];  // der Stack
  public static int x, h;
// x! soll gerechnet werden

public static void init()    // Stack einrichten
{
  h = 5;
  a[h-1] = x;
  h = h + 5;
  a[h-4] = 1;
  a[h-1] = a[h-1-5];
}

public static void rechne()
{
  a[h-5] = a[h-3];
  h = h - 5;
  a[h-3] = a[h-1] * a[h];   // Stack abbauen
}

public static void main(String arg[])
{
  x = 5;
  init();
  while (true)
  {
    if (a[h-1] == 1)
    {
```

```
      a[h-3] = 1;
      break;
    }
    else
    {
      h = h + 5;          // Stack aufbauen
      a[h-4] = 3;
      a[h-1] = a[h-5-1] - 1;
    }
  }
//  rechne();
  while (a[h-4] != 1)
    rechne();
  a[h-5] = a[h-3];
  System.out.println(x + "! = " + a[h-3]);
}
}
```

Speicherverwaltung

Als erstes betrachen wir verbundene Listen:



Jeder Eintrag enthält die Adresse des nachfolgenden Eintrags. Hier ist folgendes zu beachten:

1. Es wird zusätzlicher Speicher für die Adresse des Nachfolgers benötigt.
2. Das Streichen und Einfügen ist einfach: X[2] wird entfernt, indem der bei X[1] beginnende Pfeil auf X[3] gerichtet wird, und X[4] wird zwischen X[1] und X[2] eingefügt, indem der Pfeil von X[1] auf X[4] gerichtet wird, während er von X[4] auf X[2] zeigt.
3. Der Zugriff auf die k-te Komponente ist langsamer als bei der sequentiellen Liste. Das schadet aber dann nichts, wenn ohnehin die ganze Liste durchlaufen werden soll.
4. Das Zusammenfassen zweier Listen und die Teilung einer Liste sind leicht.

Zur Implementierung derartiger verbundenen Listen benötigt man zwei Felder, wir nennen sie INFO, INFO[p] enthält den Inhalt, der zu speichern ist, und LINK, dabei enthält LINK[p] die Adresse der nächsten Zelle. Eine Speicherplatzzuweisung realisiert man wie folgt: Man hat eine als Stack organisierte Liste NEXT der verfügbaren freien Adressen, die Variable AVAIL zeigt auf die Spitze dieser Liste. Neuen freien Speicher holt man mit

```
X:= AVAIL;   AVAIL:= NEXT(AVAIL)
```

Speicher gibt man mit

```
NEXT(X):= AVAIL;   AVAIL:= X
```

zurück.

Der Aufruf von

```
subroutine insert(item, free, position)
  info(free) = item
  next(free) = next(position)
  next(position) ) free
end
```

fügt `item` nach `position` ein.

Schließlich kann man noch doppelt verkettete Listen herstellen, wo jedes Element seinen Vorgänger und seinen Nachfolger kennt, oder ringförmig verkettete Listen, damit kann man z.B. dünn besetzte Matrizen effektiv abspeichern (Es werden (fast) nur die von Null verschiedenen Einträge gespeichert).

Wenn ein Unterprogramm aufgerufen wird, müssen aktuelle Variable des aufrufenden Programms gerettet werden. Dies betrifft insbesondere die Inhalte der Register des Prozessors, die Rücksprungadressen enthalten oder Standardaufgaben erfüllen. Dazu werden diese Werte in durchdachter Reihenfolge auf einen Stack gelegt (*push*) und zu gegebener Zeit in umgekehrter Reihenfolge zurückgeholt (*pop*). Während des Programmlaufs wird der Stack, der ja einen Teil des Arbeitsspeichers belegt, größer und kleiner, er hat aber keine „Lücken“. Zur Verwaltung des Stacks ist eine einzige Variable, der „Stack-Pointer“, nötig, deren Wert (in PCs) in einem eigens dafür reservierten Register gehalten wird.
Der vom Programmcode und vom Stack nicht genutzte Speicher steht für Programmvaiable zur Verfügung. Ein einfacher Editor reserviert sich beispielsweise ein Feld von 64 kByte (ca. 32 Druckseiten) und trägt dort die Zeichen ein, die von der Tastatur eingegeben werden. Der Speicherbedarf solcher Programme ist konstant und von vorn herein bekannt. Der Norton-Editor bekennt ggf. „Die Datei ist zu groß zum bearbeiten“.

Um Algorithmen zu implementieren, die während des Programmlaufs neue Variable konstruieren, deren Anzahl und Größe nicht vorhersehbar ist, stellen die Programmiersprachen Werkzeuge zu dem Zweck bereit, hierfür Speicherplatz zu beschaffen und zu verwalten.
Beispiele:

new, dispose, getmem, freemem	Pascal
NEW, DISPOSE, ALLOCATE, DEALLOCATE	Modula 2
malloc, free	C
$\emptyset$	Fortran 77
new	Java

Wir wollen uns an einem Beispiel klarmachen, was für Probleme da eigentlich zu bewältigen sind.

Bei einem Zeichenprogramm können Linien mit der Maus eingegeben werden. Eine Linie wird in eine Folge von Strecken zerlegt, deren jede durch vier Zahlen, die Koordinaten der Endpunkte, beschrieben werden kann. Es kann natürlich festgelegt werden, daß eine Linie aus maximal 1000 Strecken bestehen darf, aber die Anzahl der Linien soll doch nicht durch das Programm, sondern höchstens durch den verfügbaren Speicher beschränkt sein.
Für jede neue Linie ist also ein Stück freien Speichers zu finden und für die Linie zu reservieren (dieser Teil ist also „belegt“). Wenn Linien gestrichen werden, kann (soll!) der vorher belegte Speicherplatz wieder freigegeben weren.

In Pascal könnte das so aussehen:

```
type strecke = array[1..4] of integer;
     linie   = array[1..1000] of strecke;
     linienp = ^linie;         { die Anfangsadresse einer Linie }
var  lp: linienp;
begin
  ...
  new(lp);
  ...
  lp^[i][1]:= x1;
  lp^[i][2]:= y1;
  ...
  dispose(lp);
end.
```

Wenn aber viele kurze Linien gezeichnet werden sollen, ist die obige Datenstruktur unangebracht: Jede Linie verbraucht 8 kByte.

Leider verfügen einige Programmiersprachen über gar keine Möglichkeiten zur privaten Speicherverwaltung, oder der Nutzer ist mit den gelieferten Verfahren unzufrieden. Dann muß man sich eine eigene Verwaltung schaffen.

Wir beginnen mit dem einfachen Fall.

Konstante Blockgröße

Wir wollen mit verbundenen Listen arbeiten. Dazu vereinbaren wir ein sehr großes Feld `mem`, in dem wir unsere Daten unterbringen, ein Feld `avail`, das die Indizes der freien Felder enthält, ein Feld `link`, das bei jedem Listenelement auf dessen Nachfolger zeigt, und einen Zeiger `ff` (first free), der das erste freie Feld angibt. Diese Variablen sollen nicht als Parameter an Unterprogramme weitergegeben werden, sondern global gültig sein.
Es folgt eine einfache Implementation.

```
import java.io.*;
import java.lang.Math.*;

public class memory
```

```
{
  public static int nn =  10000;

  public static char a[][] = new char[nn][8];  // der Speicher
  public static int link[] = new int[nn];      // Nachfolger
  public static int avail[] = new int[nn];     // Freistellen
  public static int ff;                        // erste Freistelle
  public static char leer[] = {'#','#','#','#','#','#','#','#'};

public static void init()    //  einrichten
{
  ff = 1;
  for (int i = 1; i < nn; i++)
  {
    a[i] = (char[])leer.clone();  // nicht einfach " a[i] = leer;"
    link[i] = 0;
    avail[i] = i;
  }
  ausgabe();
}

public static int neu()
{
  int n = avail[ff];
  avail[ff] = 0;
  link[ff] = 0;
  ff++;
  return n;
}

public static int frei1(int i)
{
  a[i] = leer;
  ff--;
  int h = i;
  i = link[i];
  avail[ff] = h;
  return i;
}

public static void frei(int i)
{
  while (i > 0)
    i = frei1(i);
}
```

```
public static boolean lies(int sp)
{
  String s = new String(B.readstr());
  if (s.charAt(0) == '#')
    return false;
  for (int i = 0; i < s.length(); i++)
    a[sp][i] = s.charAt(i);
  return true;
}

public static int eingabe()
{
  int alt = -1, sp, spp;
  boolean ok;
  spp = neu();
  sp = spp;
  ok = lies(sp);
  while (ok)
  {
    alt = sp;
    link[sp] = neu();
    sp = link[sp];
    ok = lies(sp);
  }
  frei1(sp);
  link[alt] = 0;
  return spp;
}

public static void ausgabe(k)
{
  for (int i = k; i < 10; i++)
    System.out.println(i + " " + a[i] + " " +link[i] + " " + avail[i]);
}

public static void main(String arg[])
{
  init();
  spp = eingabe();
  ausgabe(spp);
}
}
```

Variable Blockgröße

Schwieriger wird es, wenn Blöcke variabler Länge zu verwalten sind.

Beispiel:
Ein Integer-Feld der Länge n kann im binären Zahlsystem als ganze Zahl interpretiert werden, deren maximaler Absolutwert $2^{16n} - 1$ ist. Stellen wir uns vor, jemand hätte Prozeduren geschaffen, die arithmetische Operationen mit solchen Zahlen realisieren. Die Summe zweier m-Bit-Zahlen ist maximal eine $(m + 1)$-Bit-Zahl, minimal aber gleich Null. Ein Produkt zweier m-Bit-Zahlen belegt maximal $2m$ Bit. Es ist also bei umfangreichen Rechnungen wenig sinnvoll, für jede lange Zahl einen Block konstanten Größe zu reservieren. Somit sind an eine Speicherverwaltung folgende Aufgaben zu stellen:

1. Speicheranforderung: Für eine neu zu schaffende Variable soll ein freier (zusammenhängender) Speicherblock gefunden werden.

2. Blockverschmelzung: Benachbarte freie Blöcke sollen zu einem größeren verschmolzen werden.

3. Speicherbereinigung (garbage collection): Nicht mehr benötigte Blöcke sollen als frei erkannt und wieder zur Verfügung gestellt werden.

Wir vereinbaren wieder ein größes Feld `memo` aus „Zellen“ konstanter Länge, ein anzufordernder Block ist dann ein zusammenhängender Bereich `memo(i:j)`. Wir haben weiter eine Liste `BEL` der belegten und eine Liste `FREI` der freien Blöcke, wir vermerken dort jeweils den Anfangsindex und die Größe des Blocks.

Methoden der Speicherplatzanforderung:

1. first fit

   Wir suchen in der Frei-Liste den ersten Eintrag, in den der gewünschte Block hineinpaßt.

   Vorteil: Man muß nicht die gesamte Liste durchlaufen.

   Nachteil: Von großen freien Blöcken wird evtl. nur wenig genutzt.

2. best fit

   Wir suchen den kleinsten Block in der Frei-Liste, in den der gewünschte Block hineinpaßt.

   Vorteil: Der Speicher wird bestmöglich genutzt.

   Nachteil: Evtl. lange Suchzeiten.

3. Blockspaltung:

   Wir verwenden eine der vorigen Methoden, melden aber den nicht verbrauchten Rest als frei.

   Nachteil: Nach einger Zeit haben wir viele kleine freie Blöcke, in die nichts mehr hineinpaßt (dies ist bei best fit besonders zu erwarten). (Schweizer Käse).

4. rotating first fit

   Wir organisieren die Frei-Liste zyklisch und suchen nicht vom Anfang, sondern von der zuletzt vergebenen Stelle aus.

Blockverschmelzung:

Um die Fragmentierung des Speichers zu begrenzen, ist es sinnvoll, bei der Freigabe eines Blocks in den beiden Nachbarblöcken nachzusehen, ob einer davon frei ist, und den vereinigten großen Block zurückzugeben. Dazu sollte `FREI` als doppelt verkettete Liste organisiert werden.

9 Ein paar Beispiele

Die folgenden Beispiele sind dem Buch von Solymosi entnommen.

Das Halteproblem

Es gibt Routinen, die nach endlicher Zeit ein Ergebnis liefern (und halten), andere tun dies nicht.

```
void haltWennGroesser1(int n)
{
  while (n != 1)
    n = n/2;
}

void evtl(int n)
{
  while (n != 1)
    if (n % 2 == 0)
      n = n/2;
    else
      n = 3*n + 1;
}
```

Wir suchen einen Algorithmus, der folgendes entscheidet:
Die Eingabe bestehe aus zwei Zeichenketten `programm` und `eingabe`. Es soll `true` ausgegeben werden, wenn `programm` ein Java-Quelltext ist und (nach der Übersetzung) `programm eingabe` hält, andernfalls (wenn es kein Java-Text vorliegt oder das Programm nicht hält) soll `false` ausgegeben werden.

Behauptung: Es gibt keine Implementierung der obigen Funktion.

Um dies zu beweisen, nehmen wir an, es gäbe eine solche Implementierung, etwa `boolean turing(String programm, String eingabe)`.
Dann implementieren wir die folgende Methode:

```
void programm(String s)
{
  if (turing(s,s))
    while (true);       // Endlosschleife
  else
    ;                   // Halt
}
```

Wir rufen Sie wie folgt auf:

```
program("void program(String s){ if turing(s,s)) while (true); else ;}");
```

1. Fall: Wenn Turing mit `true` antwortet, so kommt das Programm in eine Endlosschleife, also ist die Antwort falsch.
2. Fall: Wenn die Antword `false` ist, so hält das Program an, also ist die Antwort falsch.

Maximale Teilsummen

Gegeben ist eine Folge von n positiven und negativen Zahlen a_i, gesucht ist die Teilfolge aufeinanderfolgender Glieder, wo

$$a_k + a_{k+1} + \ldots + a_k$$

maximal ist.

1. Lösung (naiv):

```
max = 0;
for (von = 0; von <= n; von ++)
  for (bis = von; bis <= n; bis++)
  {
    s = o;
    for (i = von; i <= bis; i++)
      s = s + a[i];
    max = Math.max(s, max);
}
```

Da drei for-Schleifen zu durchlaufen sind, werden etwa n^3 Schritte nötig sein.

2. Lösung: Zeit für Raum ($O(n^2)$):

Wir sammeln in einer Tabelle schrittweise Teilsummen der Länge k, die wir aus Teilsummen der Länge $k-1$ berechnen: zuerst alle von 0 bis 0, dann von 0 bis 1, von 0 bis 2 usw. Wir setzen
`s[von][bis] = s[von - 1][bis] - a[von - 1]`
und suchen dann das Maximum.

3. Lösung: Teilen und Herrschen ($O(n \log n)$)

Wir teilen die Folge in der Mitte, dann befindet sich die maximale Teilfolge an einer von drei stellen:

Wir müssen also sowohl maximale Teilsummen suchen, die innerhalb der Folge liegen, also auch welche, die am linken bzw. rechten Rand anstoßen. Wenn dies für beide Teile geschen ist, erhalten wir die maximale Teilsumme der gesamten Folge als `max(Innen(links), innen(rechts), rechterRand(links) + linkerRand(rechts))`. Aber auch hier wird wie oben jedes Element mehrfach angefaßt. Besser als $O(n)$ kann es nicht gelingen, aber wir können diese Größenordnung erreichen.

4. Lösung:

Die maximale Teilsumme liegt am rechten Rand einer gewissen links beginnenden Teilfolge.

```
for (i = 0; i <= n; i++)
{
  randMax = Math.max(0, randMax + a[i]);
  max = Math.max(max, randmax);
}
```

10 Sortieren

Sei M eine Menge, der eine Relation $\leq$ gegeben ist. Für folgende Eigenschaften führen wir Namen ein:

1. Reflexivität: Für alle $a \in M$ gilt $a \leq a$.
2. Antisymmetrie: Aus $a \leq b$ und $b \leq a$ folgt $a = b$.
3. Transitivität: Aus $a \leq b$ und $b \leq c$ folgt $a \leq c$.
4. Vergleichbarkeit: Für $a, b \in M$ gilt $a \leq b$ oder $b \leq a$.

Eine Relation $\leq$ heißt Ordnung (manchmal auch Halbordnung), wenn die Eigenschaften 1 bis 3 erfüllt sind, sie heißt totale Ordnung (manchmal auch Ordnung), wenn zusätzlich 4. gilt. Beispiele für Ordnungen (welcher Art ?):

1. Wir betrachten die Mengen $\mathbf{N}$, $\mathbf{Z}$ oder $\mathbf{R}$ mit der gewöhnlichen Kleiner-Gleich-Relation.
2. Wir betrachten die Menge der natürlichen Zahlen und wählen als Vergleichsrelation die Teilbarkeitsrelation.
3. Sei $M = P(X)$ die Menge aller Teilmengen der Menge X, als Vergleichsrelation nehmen wir die Inklusion $\subseteq$.

4. Sei $M = X^*$ die Menge aller Worte über dem (geordneten) Alphabet X, wir ordnen sie wie im Wörterbuch:

$$x_1 \dots x_n < y_1 \dots y_m \text{ gdw. } x_i = y_i \text{ für alle } i = 1, \dots, k-1 \text{ und } x_k < y_k$$

Man nennt dies die lexikographische Ordnung.

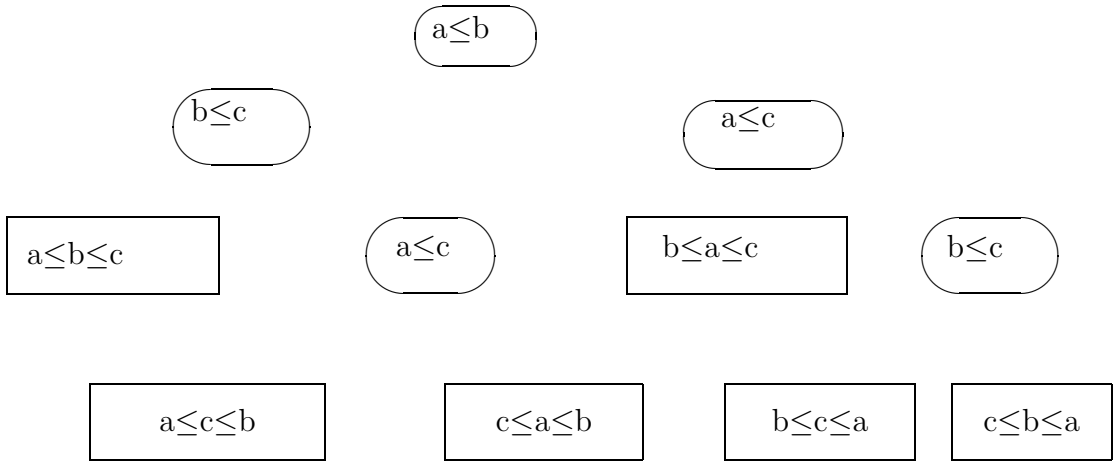
Das Sortierproblem besteht nun darin, zu einer gegebenen Eingabemenge $\{a_1, \dots, a_n\} \subset M$ eine Ausgabefolge $(a_{i_1}, \dots, a_{i_n})$ zu bilden, für die $a_{i_p} \leq a_{i_q}$ für $p < q$ gilt.

Die häufigsten Anwendungen sind:

- Das Zusammenbringen zusammengehöriger Dinge,
- das Finden gleicher Elemente,
- das Suchen eines bestimmten Elements.

Das Suchen ist in geordneten Mengen leichter als in ungeordneten.

Ein Sortieralgorithmus kann mit Hilfe eines binären Entscheidungsbaums dargestellt werden. Wir betrachten das Beispiel $M = \{a, b, c\}$.



Die minimale Anzahl der zum Sortieren notwendigen Vergleiche ist gleich der Tiefe des Entscheidungsbaums

Satz 10.1 *E_n sei ein binärer Entscheidungsbaum, um n Elemente zu sortieren. Dann ist die Tiefe von E_n mindestens $O(n \cdot \log n)$. (D.h. es gibt eine Konstante k, so daß die Tiefe mindestens gleich $k \cdot n \cdot \log n$ ist.*

Beweis: Die Anzahl der Blätter ist mindestens $n!$, da es ja soviele Anordnungsmöglichkeiten für n Elemente gibt. Ein binärer Baum der Tiefe d hat aber höchstens 2^d Blätter, also gilt für $n \geq 4$

$$2^d \geq n!,\ d \geq \log(n!),$$

$$n! \geq n \cdot (n-1) \cdot \ldots \cdot n/2 \geq \left(\frac{n}{2}\right)^{n/2}$$

$$\log(n!) \geq \frac{n}{2} \cdot \log(\frac{n}{2}) = \frac{n}{2} \cdot (\log n - 1) \geq \frac{n}{4} \log n = O(n \log n) \square$$

Wir kommen nun zu einigen speziellen Sortiervervahren.
Im Band 3 des Buchs von D. Knuth werden etwa 25 Sortierverfahren diskutiert. Einige davon sollen hier vorgestellt und später im Praktikum implementiert werden können. Es sollen jeweils N Objekte $R_1, \ldots, R_N$ sortiert werden.

10.1 Sortieren durch Zählen

Die Idee ist folgende: Der j-te Eintrag in einer geordneten Liste ist größer als genau $j-1$ andere Einträge (Wenn R größer als 27 andere ist, so muß R an Stelle 28 stehen). Wir vergleichen also jedes Objekt genau einmal mit jedem anderen und zählen, wie oft es das größere ist. In einem Feld `count` speichern wir die neuen Stellen, die die Rs einnehmen; die neue Stelle von R_j soll count[j] + 1 sein.

1. count[i]:= 0 für alle i.
2. Für $i = N,\ N-1, \ldots, 2$ gehe zu 3.
3. Für $j = i-1,\ i-1, \ldots, 1$ gehe zu 4.
4. Wenn $R_i < R_j$ ist, so setze count[j] = count[j] + 1, sonst setze count[i] = count[i] + 1.

Die benötigte Zeit ist proportional zu N^2. Wir bemerken, daß die Objekte eigentlich gar nicht sortiert wurden, aber in `count` steht, wie sie richtig angeordnet werden müssen.

10.2 Sortieren durch Verteilen (Vorsortieren)

Wir sortieren die R_i anhand ihrer „Schlüssel“ K_i. Wir setzen voraus, daß für den Schlüssel K_i von R_i und $1 \leq j \leq N$ gilt $u \leq K_i \leq v$.

1. `count(u) = 0, count(u+1) = 0, ... , count(v) = 0`
2. Für $j = 1, \ldots, N$:

   `count(`K_j`) = count(`K_j`)+1`

   Jetzt ist `count(i)` gleich der Zahl der Schlüssel, die gleich i sind.
3. Für `i = u+1, ... , v`:

   `count(i) = count(i) + count(i-1)`

   Jetzt ist `count(i)` gleich der Zahl der Schlüssel $\leq i$, insbesondere ist `count(v) = N`.
4. Für $j = N, N-1, \ldots, 1$: (Sammeln zusammengehöriger Sätze)

   `i = count(`K_j`),` $S_i = R_j$`, count(`K_j`) = i-1`

   Hier werden der Reihe nach die meistgefragtesten Objekte ausgesucht. Die Folge $S_1, S_2, \ldots$ ist nun vorsortiert.

(Datei distsort.for ?)

10.3 Sortieren durch Einfügen

Wir gehen davon aus, daß die Objekte $R_1, \ldots, R_{j-1}$ bereits an der richtigen Stelle stehen, bevor R_j bearbeitet wird. Nun vergleichen wir R_j mit $R_{j-1}, R_{j-2}, \ldots$, bis wir die Stelle finden, wo es hingehört: zwischen R_i und R_{i+1}. Dann wird von der Stelle $i+1$ alles hochgehoben und R_j anstelle von R_{i+1} eingefügt.
Man kann das Vergleichen und das Verschieben kombinieren, indem man R_j mit jedem größeren vertauscht; es sinkt dann bis zur richtigen Stelle.

1. Für $j = 2, 3, \ldots, N$ gehe zu 2.
2. $i := j - 1,\ R := R_j$.
3. Wenn $R \geq R_i$ ist, gehe zu 5.
4. $R_{i+1} := R_i;\ i := i - 1;$, wenn $i > 0$ ist, gehe zu 3.
5. $R_{i+1} := R$.

Der Zeitbedarf ist ebenfalls ungefähr N^2.
Eine Variante wäre folgende: Wenn Vergleichsoperationen langsam sind, so findet man durch binäres Suchen schnell die richtige Einfügestelle und verschiebt danach den Rest.

10.4 Sortieren durch Verketten

Eine verkettete Liste vereinfacht das Einfügen, hier sind keine Verschiebungen nötig. Wenn die zu sortierenden Objekte sequentiell in einem Feld gespeichert sind, so baut man sich während des Sortierens eine Verkettungsliste auf.
Dies kann wie folgt geschehen: Neben den zu sortierenden Objekten $R_1, \ldots, R_N$ nehmen wir noch ein künstliches R_0 hinzu. Wir verwenden eine zirkuläre Liste $L_0, \ldots, L_N$ mit $L_0 = N,\ L_N = 0$. Wenn p die Permutation mit $R_{p(1)} \leq \ldots \leq R_{p(N)}$ ist, so wird $L_0 = p(1), L_{p(i)} = p(i+1),\ L_{p(N)} = 0$ sein.

1. Führe die Schritte 2 bis 5 für $j = N - 1,\ N - 2, \ldots, 1$ aus.
2. $p := L_0,\ q := 0,\ R := R_j$.

   Als nächstes wird für R_j seine richtige Stelle in L gefunden, indem R mit den vorangehenden Einträgen verglichen wird; dabei zeigen p und q auf die aktuellen Plätze der Liste; es ist $p = L_q$, d.h. q liegt vor p.
3. Wenn $R < R_p$ ist, gehe zu 5 (dann ist $R_q < R < R_p$).
4. $q := p;\ p := L_q$. Wenn $p > 0$ ist, gehe zu 3.

   Die Zeiger p, q werden weitergesetzt. Wenn $p = 0$ ist, so ist R der größte bisher gefundene Eintrag, gehört also ans Ende der Liste: zwischen R_q und R_0.

5. $L_q := j,\ L_j := p.$

   R_j wird an der richtigen Stelle eingefügt.

Welcher Aufwand ist nötig? (N^2)

Verfeinerung:

Die Schlüssel mögen zwischen 1 und K liegen; wir teilen das Intervall in M Teilintervalle

$$[1 \dots K/M], [K/M + 1 \dots 2K/M], \dots, [(M-1)K/M \dots K]$$

und legen für jedes Intervall eine Verkettungsliste L_i an. Wenn der Schlüssel K_i ins Intervall $[rK/M...]$ paßt, verwalten wir ihn mit der Liste L_r nach dem obigen Algorithmus. Wenn die Schlüssel halbwegs gleich verteilt sind, werden die Listen deutlich kürzer.

10.5 Sortieren durch Tauschen, (bubble sort)

Dies ist eigentlich das einfachste Sortierverfahren: Von oben beginnend vertauschen wir jeweils zwei Objekte, wenn sie nicht in der richtigen Reihenfolge stehen. Die großen Elemente steigen dann wie Blasen nach oben (daher der Name).

1. Setze $Schranke = N$ (die höchste Stelle, wo die Liste nocht nicht geordnet ist).
2. $t := 0$, für $j = 1, 2, \dots, Schranke - 1$ gehe zu 3. Gehe danach zu 4.
3. Wenn $R_j > R_{j+1}$, vertausche R_j und R_{j+1} und setze $t = j$.
4. Wenn $t = 0$ ist, so sind wir fertig. Andernfalls setze $Schranke = t$ und gehe zu 2.

Welcher Aufwand ist nötig? (N^2) Alles oberhalb des zuletzt bewegten Elements ist schon in der richtigen Reihenfolge und muß nicht mehr beachtet werden.

Um die Anzahl der notwendigen *Vertauschungen* bei bubble sort beurteilen zu können, machen wir einen Ausflug:

10.6 Partitionen und Inversionstabellen

Der Folge $a = (a_1, \dots, a_n)$ mit $\{a_1, \dots, a_n\} = \{1, \dots, n\}$ (also einer Permutation) ordnen wir ihre „Inversionstabelle“ $b = (b_1, \dots, b_n)$ zu, wobei $b_j =$ Zahl der $a_i > j$ ist, die in a links von j stehen, dies ist gleich der Zahl der Inversionen der Form (x, j).

Beispiel:

$$a = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 5 & 9 & 1 & 8 & 2 & 6 & 4 & 7 & 3 \end{pmatrix}$$

Die Inversionen sind (5,1), (9,1), (5,2), (9,2), (8,2), ..., also

$$b = (2\ 3\ 6\ 4\ 0\ 2\ 2\ 1\ 0).$$

Es gilt

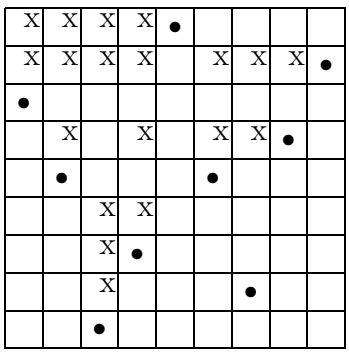
$$\begin{array}{ccccc} 0 & \leq & b_1 & \leq & n-1 \\ 0 & \leq & b_2 & \leq & n-2 \\ & \ldots & & & \\ 0 & \leq & b_{n-1} & \leq & 1 \\ & & b_n & = & 0 \end{array}$$

Wenn eine Folge b mit dieser Eigenschaft gegeben ist, so existiert eine eindeutig bestimmte Permutation a mit b als Inversionstabelle. Wir finden diese, indem wir sukzessive die relative Stellung von $n, n-1, n-2, \ldots$ bestimmen.

Wir führen das beispielhaft mit dem obigen b durch:

1. Schreibe 9
2. wegen $b_8 = 1$ steht 8 hinter 9: 9 8
3. $b_7 = 2 \Rightarrow$ 7 hinter 9, 8: 9 8 7
4. $b_6 = 2 \Rightarrow$ 6 hinter 2 der Ziffern: 9 8 6 7
5. $b_5 = 0 \Rightarrow$ 5 ganz links: 5 9 8 6 7
6. $b_4 = 4 \Rightarrow$ 4 an 5. Stelle: 5 9 8 6 4 7
7. $b_3 = 6 \Rightarrow$ 3 ganz rechts: 5 9 8 6 4 7 3
8. $b_2 = 3 \Rightarrow$ 5 9 8 2 6 4 7 3
9. $b_1 = 2 \Rightarrow$ 5 9 1 8 2 6 4 7 3

Nützlich ist eine bildliche Darstellung: In einem $n \times n$-Schachbrett zeichnen wir in der i-ten Zeile eien Punkt in Spalte a_i. An der Stelle (i, j) machen wir ein Kreuz, wenn *darunter* und *rechts daneben* ein Punkt steht.



Die Zahl der Kreuze in der Spalte j ist gleich b_j; die Gesamtzahl der Kreuze ist gleich der Zahl aller Inversionen.

Zum transponierten Feld gehört die inverse Permutation a^{-1}; da die Anzahl der Kreuze gleichbleibt, hat a^{-1} genauso viele Inversionen wie a.

Es folgt ein Algorithmus zur Bestimmung der Inversionstabelle in $n \cdot log(n)$ Schritten:

Start: $b_1 = \ldots = b_n = 0$
für $k = log(n), log(n-1), \ldots, 0$:
setze $x_s = 0$ für $0 \leq s \leq n/(2^{k+1})$
für $j = 1, \ldots, n$:
  $r = a_j/2^k \; mod 2$
  $s = a_j/2^{k+1}$
  wenn $r = 0: \; b_{a_j} = b_{a_j} + x_s$
  wenn $r = 1: \; x_s = x_s + 1$

Bei bubble sort sind so viele Vertauschungen nötig, wie die entsprechende Permutationen Inversionen besitzt. Bei einem Paß von bubble sort verringert sich jeder von 0 verschiedene Eintrag in der Inversionstafel um 1.

Eine Verbesserung der Laufzeit ist beim cocktail shaker sort gegeben: Wir durchlaufen die Folge abwechselnd auf- und abwärts.

Wenn wir weniger als $O(N^2)$ aufwenden wollen, dürfen wir uns nicht auf das Vertauschen benachbarter Objekte beschränken. Die leistet der folgende Algorithmus von Batcher:

1. Sei t die kleinste Zahl mit $2^t \geq N$; setze $p = 2^t$.
2. $q = 2^{t-1}, \; r = 0, \; d = p$
3. für alle , $0 \leq i < N$ mit `i IAND p = r`:
   (Die Funktion IAND ergibt das *bitweise* AND zweier Zahlen.)
4. wenn $K_{i+1} > K_{i+d+1}$: tausche R_{i+1} und R_{i+d+1};
5. wenn $q \neq p$: setze $d = q - p, \; q = q/2, \; r = p$ und gehe zu 3.
6. $p = p/2$, wenn $p > 0$, so gehe zu 2.

10.7 Quicksort

Dies ist das schnellste bekannte Sortierverfahren. Der Grundgedanke ist, daß der Tausch zweier Feldelemente dann den besten Fortschritt bringt, wenn diese möglichst weit voneinander entfernt sind. Wir wählen zufällig ein Element a aus der Mitte des Feldes und laufen vom linken Rand nach rechts, bis wir ein Element R_i gefunden haben, das größer als a ist. Gleichzeitig laufen wir vom rechten Rand nach links, bis wir ein Element R_j gefunden haben, das kleiner als a ist. Nun vertauschen wir R_i und R_j und fahren fort, bis sich die Indizes getroffen haben. Nun haben wir zwei Teilfelder, deren Elemente alle kleiner (bzw. größer) als a sind. Wir haben die Aufgabe in zwei Teilaufgaben zerlegt. Im Idealfall sind beide Teilfelder gleich groß, im schlechtesten Fall (a ist ein Extremwert) ist ein Teilfeld leer, im zweiten Fall haben wir n^2 Arbeitsschritte, das ist nicht gerade „quick“. Aber im „Durchschnitt“ reichen $n \log n$ Schritte aus.

Ein einfacher, rekursiver Algorithmus ist der folgende:

Quicksort(S)
Wenn $|S| > 1$ ist, dann wähle zufällig ein $a \in S$.
Setze $S_1 = \{x \in S \mid x < a\}, S_2 = \{x \in S \mid x \geq a\}$.
Quicksort(S_1)
Quicksort(S_2)
Setze $S[i] = S_1[i]$ für $i = 1, \ldots, |S_1| = m, S[j] = S_2[j - m - 1]$ für $j = m + 1, \ldots, n$.

Einen cleveren (nichtrekursiven) Algorithmus findet man bei Knuth, Band 3, Seite 114, er stammt von C.A.R. Hoare (1962). Wir wollen dies hier kurz darstellen.

Es sollen $R_1, \ldots R_N \in \mathbf{Z}$ sortiert werden. Wir setzen $R_0 = -\infty,\ R_{N+1} = \infty$. Als das Element, das an seine endgültige Stelle gebracht wird, können wir R_1 wählen. Alle Vergleiche innerhalb eines Zyklus beziehen sich auf dieses Element; es ist vernünftig, eine Extra-Variable mit dem Wert von `R(1)` zu belegen, denn Operationen mit (normalen) Variablen sind schneller als welche mit indizierten Variablen.
Um die Grenzen von noch zu sortierenden Teilen zu speichern, genügen jeweils zwei Variable, die auf dem Stack abgelegt werden können. Dabei werden die Grenzen des jeweils größeren Teils gemerkt und der kleinere Teil wird bearbeitet. Teilstrukturen der Länge $\leq M$ werden unsortiert gelassen, damit wird am Ende aufgeräumt.

1. Wenn $N \leq M$, gehe zu 9. Sonst: Erzeuge leeren Stack, $l := 1,\ r := N$.

2. $i := l;\ j := r + 1, R := R_l$. (Neuer Durchgang: Wir sortieren nun $R_l, \ldots, R_r$, es gilt $r \geq l + M,\ R_{l-1} \leq R_i \leq R_{r+1}$ für $i = l, \ldots, r$).

3. (Vergleiche R_i mit R) $i := i + 1$, solange $R_i < R$ ist.

4. (Vergleiche R mit R_j) $j := j - 1$, solange $R < R_j$ ist.

5. Wenn $j \leq i$ ist, so vertausche R_l mit R_j und gehe zu 7. (Die Indizes haben sich getroffen.)

6. Vertausche R_i und R_j und gehe zu 3.

7. Wenn $r - j \geq j - l > M$, so lege $(j + 1,\ r)$ auf den Stack, setze $r = j - 1$, gehe zu 2.

   Wenn $j - l > r - j > M$ ist, so lege $(l,\ j - 1)$ auf den Stack, setze $l = j + 1$, gehe zu 2.

   Wenn $r - j > M \geq j - l$ ist, setze $l = j + 1$, gehe zu 2.

   Wenn $j - l > M \geq r - j$ ist, setze $r = j - 1$, gehe zu 2.

8. Wenn der Stack nicht leer ist, so hole $(l,\ r)$ vom Stack und gehe zu 2.

9. (Nacharbeit, falls $M > 1$ ist) Für $j = 2, \ldots, N$ gehe folgende Schritte: $R := R_j,\ i := j - 1$, wiederhole $R_{i+1} := R_i,\ i := i - 1$, bis $R_i \leq R$ ist. Setze $R_{i+1} = R$. (Wir lassen R_j sinken.)

Das folgende Programm realisiert diesenAlgorithmus.

```
import java.io.*;

public class quick
{
  public static int nn =  20;
  public static int stack[] = new int[nn+1];
  public static int rr[] = new int[nn+1];
  public static int i, j, r, l, st, rh, hh;

public static void init()    //  einrichten
{
  random.h = 32003;
  for (int i = 1; i < nn-1; i++)
    rr[i] = random.f2n(15, i);
  rr[0] = -100000; rr[nn-1] = 100000;
}

public static void s1()
{
  l = 1; r = nn - 1; st = 0;
}

public static void s2()
{
  i = l; j = r + 1; rh = rr[l];
  s3();
}

public static void s3()
{
  i++;
  if (rr[i] < rh)
    s3();
  s4();
}

public static void s4()
{
  j--;
  if (rh < rr[j])
    s4();
  s5();
}
```

```
public static void s5()
{
  if (j <= i)
  {
    hh = rr[l]; rr[l] = rr[j]; rr[j] = hh;
    s7();
  }
  else
    s6();
}

public static void s6()
{
  hh = rr[i]; rr[i] = rr[j]; rr[j] = hh;
  s3();
}

public static void s7()
{
  if ((r-j >= j-l) && (j-l > 1))
  {
    stack[st+1] = j + 1;
    stack[st+2] = r;
    st = st + 2;
    r = j - 1;
    s2();
  }
  else if ((j-l > r-j) && (r-j > 1))
  {
    stack[st+1] = l;
    stack[st+2] = j - 1;
    st = st + 2;
    l = j + 1;
    s2();
  }
  else if ((r-j > 1) && (1 >= j-l))
  {
    l = j + 1;
    s2();
  }
  else if ((j-l > 1) && (1 >= r-j))
  {
    r = j - 1;
    s2();
```

```
  }
  else s8();
}

public static void s8()
{
  if (st > 0)
  {
    r = stack[st];
    l = stack[st-1];
    st = st - 2;
    s2();
  }
}

public static void main(String arg[])
{
  init();
  s1();  s2();  s3();  s4();  s5();  s6();  s7();  s8();
  System.out.println();
  for (int i = 1; i < nn; i++)
    System.out.print(rr[i] + " ");
}
}
```

10.8 Binärdarstellung der Schlüssel

Dies ist eine Variante von quick sort, die die Bitdarstellung der Schlüssel (-Zahlen) nutzt. Wir sortieren nach dem höchstwertigen Bit: zuerst alle mit 0, dann alle mit 1. also:

Suche den am weitesten links stehenden Schlüssel K_i mit führendem Bit = 1;
Suche den am weitesten rechts stehenden Schlüssel K_j mit führendem Bit = 0.
Tausche R_i und R_j, erhöhe i, senke j, bis $i > j$ ist.
Sei $F_0 = \{$ Schlüssel mit führendem Bit = 0 $\}$,
sei $F_1 = \{$ Schlüssel mit führendem Bit = 1 $\}$.
Num Sortieren wir F_0 und F_1 durch Vergleich des zweiten Bits, usw.

Im Einzelnen ist folgendes zu tun: (Wenn $\max(K_i) < 2^m$ ist, so brauchen wir einen Stack der Größe $m - 1$ für m-Bit-Zahlen.

1. leeren Stack erzeugen, $l = 1,\ r = N,\ b = 1$ (links, rechts, Bitnummer)

2. Wenn $l = r$ ist, so gehe zu 10. Sonst setze $i = l,\ j = r$ (Neuer Durchgang: wir untersuchen das Bit b bei $K_l, \ldots, K_r$).

3. Wenn Bit b von K_i gleich 1 ist, gehe zu 6.

4. $i = i + 1$; wenn $i \leq j$: gehe zu 3
   sonst gehe zu 8
5. Wenn Bit b von k_{j+1} gleich 0 ist, gehe zu 7
6. $j = j - 1$; wenn $i \leq j$: gehe zu 5
   sonst gehe zu 8
7. Tausche R_i und R_{j+1}, gehe zu 4
8. (Ein Durchgang ist beendet, es ist $i = j + 1$, Bit b von $K_l, \ldots, K_j$ ist 0,Bit b von $K_i, \ldots, K_r$ ist 1.)
   $b = b + 1$; wenn $b > m$: gehe zu 10
   Wenn $j < l$ oder $j = r$: gehe zu 2 (Alle b-Bits waren gleich)
   Wenn $j = l$, setze $l = l + 1$, gehe zu 2 (nur ein b-Bit ist 0)
9. Lege das Paar (r, b) auf den Stack (wir merken uns die rechte Grenze, wo noch Bit b geprüft werden muß), setze $r = j$; gehe zu 2
10. (Stack leeren) Wenn der Stack leer ist, sind wir fertig. Sonst setze $l = r + 1$, hole (r', b') vom Stack, setze $r = r',\ b = b'$ und gehe zu 2.

10.9 Sortieren durch direkte Auswahl

Wir suchen den kleinsten Schlüssel, geben den Entsprechenden Datensatz aus und ersetzen den Schlüssel durch ∞. Wir wiederholen dies, bis N Sätze bearbeitet sind.

Also: Die zu sortierende Liste muß vollständig vorliegen, die Ausgabe erfolgt sequentiell. Dies ist genau das umgekehrte Vorgehen wie beim Sortieren durch Einfügen: dort ist die Eingabe sequentiell, Ergebnisse erhält man erst ganz zum Schluß.

Bei jedem Durchgang sind $N - 1$ Vergleiche nötig. Besser ist es, den ausgewählten Satz an „seine“ Stelle zu bringen (Tausch mit dem Inhaber) und später nicht mehr zu betrachten:

Für $j = N, N - 1, \ldots, 2$:
  sei $K_i = \max(K_1, \ldots, K_j)$
  Tausche R_i und R_j. (dann ist $R_j, \ldots, R_N$ geordnet).

Sind für die Bestimmung des Maximums von N Zahlen wirklich $N-1$ Vergleiche nötig, ist die Komplexität also immer noch $O(N^2)$?

Besser: Wir zerlegen die Daten in $\sqrt{N}$ Gruppen zu $\sqrt{N}$ Elementen, suchen in jeder Gruppe das Maximum und bestimmen das Maximum der Gruppenersten; dieses wird dann entfernt. Dazu sind etwa $\sqrt{N} + \sqrt{N}$ Vergleiche nötig.

Wenn das iteriert wird, können wir in $O(N \cdot \sqrt{N})$ Schritten sortieren. Man nennt dieses Verfahren „quadratische Auswahl“. Es ist auch eine kubische Auswahl mit $O(N \cdot \sqrt[3]{N})$ Schritten denkbar, wir nähern uns langsam der magischen Schranke $O(n \cdot log(n)$.

10.10 tree selection

Diese Schranke wird wie bei Tennisturnieren im k.o.-System erreicht. Wenn 2^n Kandidaten antreten, ist der Sieger nach n Runden ermittelt. Wer aber ist der Zweitbeste? Es ist einer der n Spieler, der gegen den Sieger verloren hat.

Wir belegen die Blätter eines binären Baums mit N Blättern durch die Schlüssel K_i und belegen jeweils den Vaterknoten durch das Maximum der beiden Söhne. In der Wurzel haben wir also den Ersten E. Um den zweiten zu finden, ersetzen wir E überall durch $-\infty$ und bilden entlang der Astfolge zur Wurzel wieder die Maxima, die Länge dieses Weges ist gleich $log(N)$ und wir haben den Zweiten gefunden; usw. Insgesamt sind also $N \cdot log(N)$ Vergleiche nötig.
Dies ist das Peter-Prinzip: Jeder steigt in der Hierarchie so weit auf, bis er sein Nineau der Inkompetenz erreicht hat.

10.11 heap sort

Eine Folge $K_1, \ldots, K_N$ heißt Halde (heap), wenn $K_{i/2} \geq K_i$ für alle i ist, also

$$K_1 \geq K_2,\ K_1 \geq K_3,\ K_2 \geq K_4, \ldots$$

Demnach ist $K_1 = \max(K_1, K_2, \ldots, K_N)$, das größte Element liegt oben auf der Halde. Man veranschaulicht sich das am besten an einem binären Baum: der Vater ist größer als die beiden Söhne.

Es gelte

$$K_{j/2} \geq K_j \text{ für } l < j/2 < j \leq N;$$

dies ist für $l = N/2$ trivialerweise erfüllt (es gibt kein j). Diese Heap-Bedingung wollen wir schrittweise für kleinere j erhalten. Wir formen die Eingabeliste zu einer Halde um, entfernen die Spitze und bringen sie an die richtige Stelle.

1. setze $l = N/2 + 1,\ r = N$

2. (l oder r verkleinern)

   wenn $l > 1$ ist, setze $l = l - 1,\ R = R_l,\ K = K_l$

   sonst setze $R = R_r,\ K = K_r,\ R_r = R_1,\ r = r - 1$

   Wenn nun $r = 1$ ist, setze $R_1 = R$ und beende.

3. (Wir haben eine Halde ab l und R_k hat für $r < k \leq N$ den richtigen Platz; wir wollen eine Halde ab k mit $k = l/2$.)

   setze $j = l$

4. (ab hier ist $i = j/2$)

   Setze $i = j,\ j = 2i$

   wenn $j < r$ ist, gehe zu 5

   wenn $j = r$ ist, gehe zu 6

   wenn $j > r$ ist, gehe zu 8

5. wenn $K_j < K_{j+1}$ setze $j = j + 1$ (finde größeren Sohn)
6. wenn $K \geq K_j$ ist, gehe zu 8
7. (hochheben) setze $R_i = R_j$, gehe zu 4
8. (speichere R) setze $R_i = R$, gehe zu 2

Die Komplexität ist garantiert $N \cdot log(N)$, denn in Schritt 4 wird das Intervall halbiert.

10.12 Sortieren durch Mischen (merge sort)

Hier haben wir zwei sortierte Folgen, die zu einer Folge vereinigt werden sollen. Wir verleichen die beiden Minima, geben das kleinere aus, entfernen es und beginnen von vorn. Wenn eine der Listen leer ist, so ist die andere Liste der Rest.

Seien also

$$x_1 \leq x_2 \leq \ldots \leq x_m \text{ und } y_1 \leq y_2 \leq \ldots \leq y_n$$

gegeben.

1. $i = 1,\ j = 1,\ k = 1$
2. wenn $x_i \leq y_j$, so gehe zu 3
   sonst gehe zu 5
3. setze $z_k = x_i,\ k = k + 1,\ i = i + 1$
   wenn $i \leq m$ ist, gehe zu 2
4. Setze $(z_k, \ldots z_{m+n}) = (y_j, \ldots y_n)$ und beende.
5. setze $z_k = y_j,\ k = k + 1,\ j = j + 1$
   wenn $j \leq n$ ist, gehe zu 2
6. Setze $(z_k, \ldots, z_{m+n}) = (x_i, \ldots, x_m)$ und beende.

Der Aufwand ist $m + n$, d.h. Mischen ist leichter als Sortieren. Dies ist eines der ersten auf Rechnern implementiertes Sortierverfahren (J. v. Neumann, 1945).

10.13 Natürliches Mischen

Wir schauen uns die Eingabefolge von beiden Seiten an. Ein „Aufstieg“ ist eine Folge

$$a_i \leq a_{i+1} \leq \ldots \leq a_k.$$

Die Gesamtfolge ist eine Folge von Aufstiegen, die je durch einen Abwärtsschritt getrennt werden. Wie gesagt betrachten wir die Folge von links und von rechts.

Beispiel (die markierten Abschnitte stellen jeweils Aufstiege dar):

$$\underbrace{503 \quad 703 \quad 765}\ \underbrace{61 \quad 612 \quad 908}\ \underbrace{154 \quad 275 \quad 426 \quad \overbrace{653}}\ \overbrace{897 \quad 509 \quad 170}\ \overbrace{677 \quad 512 \quad 87}$$

Wir mischen nun den am weitesten links stehenden Aufstieg mit dem am weitesten rechts stehenden Aufstieg zu einem neuen linken Aufstieg und entfernen die alten. Dann mischen wir die beiden nächsten Aufstiege zu einem neuen Aufstieg rechts, usw.

Das Ergebnis des ersten Durchgangs ist in unserem Beispiel

$$\underbrace{87 \quad 503 \quad 512 \quad 677 \quad 703 \quad 765}\ \underbrace{154 \quad 275 \quad 426 \quad \overbrace{653}}\ \overbrace{908 \quad 897 \quad 612 \quad 509 \quad 170 \quad 61}$$

Als Eingabedaten haben wir $R_1, \ldots, R_N$; wir brauchen einen Hilfsspeicher $R_{N+1}, \ldots, R_{2N}$, dessen Anfangsinhalt irrelevant ist.

1. $s = 0$

   (Wenn $s = 0$ ist, bringen wir die Dinge von $R_1, \ldots, R_N$ nach $R_{N+1}, \ldots, R_{2N}$, bei $s = 1$ umgekehrt.)

2. wenn $s = 0$ ist, setze $i = 1,\ j = N,\ k = N + 1,\ l = 2N$

   wenn $s = 1$ ist, setze $i = N + 1,\ j = 2N,\ k = 1,\ l = N$

   $d = 1,\ f = 1$ (Ausgaberichtung; noch ein Paß?)

3. (Schritte 3 bis 7: Mischen)

   wenn $K_i > K_j$: gehe zu 8

   wenn $i = j$: setze $R_k = R_i$, gehe zu 13

4. setze $R_k = R_i,\ k = k + d$

5. (Abwärtsschritt?) $i = i + 1$; wenn $K_{i-1} \leq K_i$: gehe zu 3

6. setze $R_k = R_j,\ k = k + d$

7. (Abwärts?) $j = j - 1$, wenn $K_{j+1} \leq K_j$: gehe zu 6

   sonst gehe zu 12

8. (Schritte 8 bis 11 sind dual zu 3, ..., 7) setze $R_k = R_j,\ k = k + d$

9. (Abwärts?) $j = j - 1$, wenn $K_{j+1} \leq K_j$: gehe zu 3

10. setze $R_k = R_i,\ k = k + d$

11. (Abwärts?) $i = i + 1$, wenn $K_{i-1} \leq K_i$: gehe zu 10

12. (Seiten vertauschen) Setze $f = 0, d = -d$, tausche $k \leftrightarrow l$, gehe zu 3

13. (Tausche unten und oben)

    wenn $f = 0$, setze $s = 1 - s$, gehe zu 2

    Sonst sind wir fertig. Wenn $s = 0$ sein sollte, ist noch der Hilfsspeicher nach unter zu kopieren.

Verbesserung: Wir legen fest, daß beim ersten Durchgang alle Aufstiege die Länge 1 haben (d.h. wir prüfen nicht, wo die Aufstiege enden). Dann haben aber beim zweiten Durchgang alle Aufstiege die Länge 2, beim dritten die Länge 4 ...

Leider ist der Speicheraufwand mit $2N$ recht groß, wobei stets die Hälfte des Speichres ungenutzt ist. Besser wäre es, mit verbundenen Listen zu arbeiten.

10.14 list merge sort

Wir stellen eine Link-Liste $L_1, \ldots, L_N$ bereit, wo Zahlen zwischen $-N-1$ und $N+1$ stehen können; zusätzlich nutzen wir $L_0, L_{N+1}, R_0, R_{N+1}$.

Am Ende ist L_0 der Index des kleinsten R_i und L_k ist der Index des NAchfolgers von R_k, oder $L_k = 0$, falls R_k maximal ist. R_0 und R_{N+1} sind die Listenköpfe. Wenn ein Link negavit ist, so ist das Ende einer geordneten Teilliste erreicht.

Wir verwenden die Fortran-Funktion SIGN folgendermaßen: `Ls = SIGN(p, Ls)`; dabei behält `Ls` sein Vorzeichen bei und erhält den Wert `p` oder `-p`.

1. setze $L_0 = 1,\ L_{N+1} = 2,\ L_{N-1} = L_N = 0$

   fr $i = 1, \ldots N-2:\ L_i = -(i+2)$

   (Wir haben zwei Teillisten $R_1, R_3, R_5, \ldots$ und $R_2, R_4, R_6, \ldots$ und die negativen Links bedeuten, daß jede geordnete Teilliste einelementig ist.)

2. (neuer Durchgang; s ist die letzte bearbeitete Eintragung, t ist das Ende der vorher bearbeiteten Liste, p, q durchlaufen die Liste)

   setze $s = 0,\ t = N+1,\ p = L_s,\ q = L_t$. Wenn $q = 0$ ist, sind wir fertig.

3. wenn $K_p > K_q$: gehe zu 6

4. setze $L_s = SIGN(L_s, p),\ s = p,\ p = L_p$; wenn $p > 0$: gehe zu 3

5. (Teilliste ferstigstellen)

   setze $L_s = q,\ s = t$

   wiederhole $t = q,\ q = L_q$ bis $q \leq 0$

   gehe zu 8

6. setze $L_s = SIGN(L_s, p),\ s = q,\ q = L_q$

   wenn $q > 0$: gehe zu 3

7. setze $L_s = p,\ s = t$

   wiederhole $t = p,\ p = L_p$ bis $p \leq 0$

8. (Durchgang fertig, d.h. $p \leq 0, q \leq 0$)
   setze $p = -p,\ q = -q$
   wenn $q = 0$: setze $L_s = SIGN(L_s, p),\ L_t = 0$, gehe zu 2
   sonst gehe zu 3

11 Das charakteristische Polynom

Wenn A eine $n \times n$-Matrix und x eine Unbestimmte ist, so heißt die Determinante $\chi_A(x) = \mid xE_n - A \mid$ das charakteristische Polynom von A. Da die Einträge der Matrix $xE - A$ keine Körperelemente, sondern Polynome (vom Grad 0 oder 1) sind, kann die Determinante nicht mit Hilfe des Gaußschen Algorithmus berechnet werden. Was nun?

Für die Leibnizsche Methode wäre eine Polynomarithmetik nötig; der Rechenaufwand ist mit $n!$ unakzeptabel hoch.

Wenn man die Tatsache ausnutzen will, daß der Koeefizient von x^{n-1} in $\chi_A(x)$ gleich der Summe der i-Hautminoren von A ist, so muß man bedenken, daß der Koeffizient von $x^{\frac{n}{2}}$ die Summe von $\binom{n}{\frac{n}{2}}$ Determinanten ist, allein deren Anzahl ist schon riesig.

Wenn man sich aber die Mühe macht und eine Polynomarithmetik implementiert, so geht es leicht. Man braucht die Addition/Subtraktion, Multiplikation und Division mit Rest für Polynome in einer Variablen. Wir bringen dann die Polynommatrix $xE_n - A$ schrittweise in Dreiecksform: Durch Zeilen-/Spaltentausch bringen wir die Komponente minimalen Grades an die Position (1,1). Dann dividieren wir mit Rest: $a_{i,1} = a_{1,1} \cdot q + r$. Wenn wir nun das q-fache der ersten Zeile von der i-ten subtrahieren, erhalten wir in der ersten Spalte gerade r, was Null ist oder aber zumindest von minimalem Grad (aller Komponenten) ist. Dies iterrieren wir, bis die erste Spalte Nullen enthält. Dann ist die zweite Spalte dran usw.

Drei Dinge sollen noch angeführt werden:
Mit demselben Verfahren kann man die Matrix in Diagonalform überfhren, und man kann es erreichen, daß die Diagonalkomponenten einander teilen. Diese Diagonalform ist eindeutig bestimmt und heißt Smithsche Normalform; die Diagonalkomponenten heißen ihre Invariantenteiler. Der „höchste“ Invariantenteiler von $xE_n - A$ ist das Minimalpoynom von A.
Das Ganze klappt natürlich auch für beliebige Polynommmatrizen.
Wenn die Einträge einer Matrix ganzzahlig sind, kann man mit dem beschriebenen Verfahren die Determinante ohne Divisionen berechnen.

Wir wollen den Rechenaufwand grob abschätzen:
Es entstehen Polynome bis zum Grad n. Um ein solches zu Null zu machen, sind maximal n Zeilenoperationen nötig. Eine Zeilenoperation mit Polynomen kostet etwa n^2 Rechenoperationen. Es sind etwa n^2 Komponenten zu bearbeiten. Die Zeitkomplexität ist also etwa n^5.

Ein noch besseres Verfahren basiert auf den Newtonschen Formeln [1], das darüberhinaus fast ohne Divisionen auskommt:
Für die Koeffizenten von $\chi_A(x) = \sum a_i x^{n-i}$ gilt die Rekursionsgleichung

$$a_i = -\frac{1}{i} \sum_{j=0}^{i-j} a_j \text{Spur}(A^{i-j}).$$

Also

$$a_0 = 1,$$
$$a_1 = -\text{Spur}(A),$$
$$a_2 = -\frac{1}{2}\text{Spur}((A + a_1 E)A),$$

usw.
Es sind ungefähr n^4 Rechenoperationen notwendig.
Das soeben vorgestellte Verfahren stammt von Leverrier und Faddejew. Da der konstante Term des charakteristischen Polynoms bis aufs Vorzeichen gleich der Determinante von A ist, kann diese relativ schnell, fast ohne Divisionen berechnet werden.

Nach dem Satz von Hamilton/Cayley gilt

$$\sum a_i A^{n-i} = 0,$$

Nach Multiplikation mit A^{-1} (falls dies existiert) kann diese Gleichung nach A^{-1} aufgelöst werden; die Inverse einer Matrix kann also auch fast ohne Divisionen berechnet werden. Dies wurde in den achtziger Jahren von einem Informatiker wiederentdeckt und publiziert. In einer Rezension des betreffenden Artikels wurde das Verfahren allerdings als „numerisch unstabil“, weil Divisionen erfordernd, abqualifiziert. Ich kann mir aber kein Berechnungsverfahren zur Matrixinversion vorstellen, das ohne Divisionen auskommt, nur welche mit mehr oder weniger vielen.

Daß die Matrixinversion genauso schwierig wie die Matrixmultiplikation ist, erkennen wir daran, daß sich die Multiplikation auf die Inversion zurückführen läßt:

$$\begin{pmatrix} E & A & 0 \\ 0 & E & B \\ 0 & 0 & E \end{pmatrix}^{-1} = \begin{pmatrix} E & -A & AB \\ 0 & E & -B \\ 0 & 0 & E \end{pmatrix}$$

12 Graphen

Definition: Sei E eine Menge, deren Elemente wir „Ecken“ nennen werden, und K eine Menge ungeordneter Paare (e_1, e_2) mit $e_i \in E$. Dann nennen wir das Paar $G = (E, K)$ einen Graphen. Die Elemente von K nennen wir die Kanten von G. Wir sagen, die Kante (e_1, e_2) verbindet die Ecken e_1 und e_2.
Ein Graph $G_1 = (E_1, K_1)$ heißt Teilgraph von G, wenn $E_1 \subset E$ und $K_1 \subset K$ ist.

[1] Für einen Beweis vgl. Grassmann, Algebra und Geometrie I, II, III im Kapitel „Polynome“; `http://www-irm.mathematik.hu-berlin.de/~hgrass`.

G_1 heißt spannender Teilgraph von G, wenn $E_1 = E$ ist.
Sei $x \in E$, dann bezeichnen wir mit $deg(x) = |\{y \in E \mid (x,y) \in K\}|$ den Grad der Ecke x, es ist die Anzahl der Kanten, die von x ausgehen.
Ein Graph heißt vollständig, wenn jeweils zwei seiner Ecken verbunden sind.

Lemma 12.1 $\sum_{x \in E} deg(x) = 2\,|K|\,.$

Beweis: Jede Kante verbindet zwei Ecken.

Definition: Eine Folge $x_1, \ldots, x_n$ von Ecken heißt Weg der Länge n, wenn jeweils $(x_i, x_{i+1}) \in K$ gilt. Mit $d(x,y)$ bezeichnen wir den Abstand der Ecken x, y, dies ist als die Länge des kürzesten Wegs von x nach y definiert.
Ein Graph heißt zusammenhängend, wenn es zu zwei beliebigen Ecken einen Weg gibt, der sie verbindet.
Wir nennen G einen gewichteten Graphen, wenn jeder Kante (x,y) eine positive reelle Zahl $d(x,y)$ zugeordnet ist, dies nennen wir das Gewicht der Kante und als Abstand zweier Ecken wählen wir die die Summe der Gewichte eines kürzesten Verbindungswegs.

Wir wollen nun ein Verfahren von Dijkstra (1959) behandeln, das einen kürzesten Weg zwischen zwei Ecken v und w eines Graphen bestimmt.
Dazu führen wir folgende Funktion ein: Sei $v \notin U \subset E$ eine Menge von Ecken, die v nicht enthält. Für $x \in U$ setzen wir

$$l_U(x) = \begin{cases} d(v,x), & \text{wenn es einen Weg } v = v_1, \ldots, v_n = x \text{ mit } v_i \notin U \\ & \text{für } i = 1, \ldots, n-1 \text{ gibt,} \\ \infty & \text{sonst.} \end{cases}$$

Sei nun $x_1 \in U$ die Ecke in U, für die das Minimum $l_U(x_1) = \min(l_U(x) \mid x \in U)$ angenommen wird. Dann ist $l_U(x_1)$ die Länge des kürzesten Weges von v nach x_1, denn andernfalls gäbe es einen kürzeren Weg von v nach x_1. Dieser Weg kann aber nicht vollständig (bis auf x_1) außerhalb von U liegen, da ja $l_U(x_1)$ minimal sein sollte. Dieser Weg enthält also eine Ecke z aus U und wir hätten $l_U(z) < l_U(x_1)$, ein Widerspruch. (Für die anderen Ecken x aus U kann es Wege von v aus geben, die kürzer als $l_U(x)$ sind.)
Wie kann man nun l_U bestimmen? Im folgenden Algorithmus haben wir stets die folgende Situation:

1. Für jede Teilmenge $U \subset E$ kennt man $l_U(x)$ für alle $x \in U$.
2. Für alle $y \in W = E - U$ kennt man einen kürzesten Weg von v nach y, der U nicht berührt.

Also: Man kennt für jede Ecke die Länge des kürzesten Wegs von v aus, der mit Ausnahme höchstens der letzen Ecke außerhalb von U verläuft; für die Ecken außerhalb von U sind dies überhaupt die kürzesten Wege.
Sei nun $z \in U$ beliebig, wir setzen $V = U - \{z\}$. Wenn nun $x \in U$ ist, so gilt

$$l_V(x) = \min(l_U(x), l_U(z) + d(z,x)).$$

Beweis: Von v gelangt man auf kürzestem Weg, ohne V vorzeitig zu betreten, über z oder nicht über z. Wenn wir nicht über z gehen, so wird U auch nicht betreten,

also ist $l_V(x) = l_U(x)$. Im anderen Fall ist z die vorletzte Ecke eines kürzesten Weges von v nach x, denn sonst gäbe es ein $y \notin U$ auf einem Weg zwischen v und x und wir kennen einen kürzesten Weg von v nach y, der U nicht berührt. Damit hätten wir einen kürzesten Weg, der z nicht enthält, im Widerspruch zu Voraussetzung. Also ist hier $l_V(x) = l_U(z) + d(z, x)$.
Nun kommen wir zum Algorithmus, der $d(v, x)$ bestimmt und einen Weg dieser Länge aussucht.

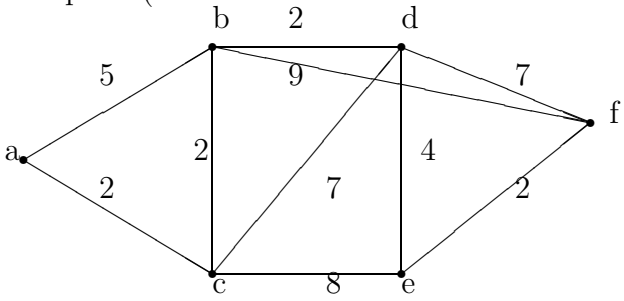
1. Initialisierung: $W = \emptyset,\ U = E - W,\ l_U(v) = 0, l_U(x) = \infty$ für $x \neq v,\ p(x) = \star$ für alle $x \in E$, dies wird später die nächste Weg-Ecke sein.

2. Sei $z \in U$ die Ecke mit minimalen l_U-Wert (im ersten Schritt ist dies v), wir setzen $d(v, z) = l_U(z)$.

3. Setze $W := W \cup \{z\}, V = U - \{z\}$. Für alle $x \in V$ setzen wir $l_V(x) = \min(l_U(x), l_U(z) + d(z, x))$ und wenn dabei $l_U(x) > l_U(z) + d(z, x)$ ist, so setzen wir $p(x) = z$. Nun setzen wir $U := V$.

4. Wenn $W = E$ ist, so sind wir fertig. Wenn $l_U(x) = \infty$ für alle $x \in U$ ist, so ist der Graph nicht zusammenhängend. Andernfalls gehe zu 2.

Der Arbeitsaufwand hat die Größenordnung $|K^2|$. Einen kürzesten Weg von x nach v geht durch die Ecken $x, p(x), p(p(x)), \ldots$.
Wir führen dies in einem Beispiel durch. Wir betrachten den folgenden gewichteten Graphen (die Gewichte sind an die Kanten herangeschrieben):



Wir suchen die kürzesten Wege zwischen a und den anderen Ecken. Bei der Initialisierung wird $W = \emptyset, U = \{a, b, c, d, e, f\}, l_U(a) = 0, l(x) = \infty$ sonst, $p(x) = \star$ für alle x.
Beim ersten Durchlauf wird
$z = a, d(a, a) = 0, W = \{a\}, U = \{b, c, d, e, f\}, l_U(b) = 5, l_U(c) = 2, p(b) = a, p(c) = a$.
Zweiter Durchlauf:
$z = c, d(a, c) = 2, W = \{a, c\}, U = \{b, d, e, f\}, l_U(b) = 4, l_U(d) = 9, l(e) = 10, p(b) = c, p(d) = c, p(e) = c$.
Dritter Durchlauf:
$z = b, d(a, b) = 4, W = \{a, c, b\}, U = \{d, e, f\}, l(d) = 6, l(f) = 13, p(d) = b, p(f) = b$.
Vierter Durchlauf:
$z = d, d(a, d) = 6, W = \{a, b, c, d\}, U = \{e, f\}$.
Fünfter Durchlauf:
$z = e, d(a, e) = 10, W = \{a, b, c, d, e\}, U = \{f\}, l(f) = 12, p(f) = e$.

Letzter Durchlauf: $z = f, d(a, f) = 12, U = \emptyset$.

Als Datenstrukturen für die Verwaltung bieten sich die folgenden an:

1. Die Adjazenzmatrix des Graphen: $A_G = (a_{ij})$ mit $a_{ij} = \begin{cases} 1, & (e_i, e_j) \in K \\ 0 & \text{sonst} \end{cases}$.

   Dies ist nicht besonders effektiv, wenn viele Nullen gespeichert werden.

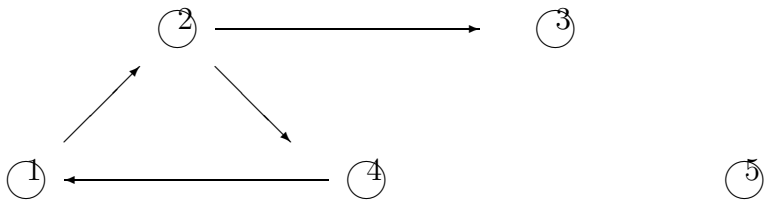
2. Wenn der Grad der Eckpunkte durch eine Zahl d beschränkt ist, so kann man die Information in einer verketteten Liste ablegen, deren einzelne Zellen folgende Daten enthalten:

   - laufende Nummer des Eckpunkts,
   - Anzahl der Nachbarn,
   - ein Feld der Größe d für die Nummern der Nachbarecken.

3. Für binäre Graphen, das sind solche, wo jeder Eckpunkt einen „Vater“ und höchstens zwei „Söhne“ hat, kann man ein Feld g mit folgenden Informationen anlegen:

   Die Söhne von $g[i]$ sind $g[2i]$ und $g[2i + 1]$, der Vater von $g[i]$ ist $g[i \text{ div } 2]$.

Welche Informationen kann man der Adjazenzmatrix eines Graphen entnehmen?



Dies ist ein „gerichteter“ Graph; wir verallgemeinern die Zuordnungsvorschrift für die Adjazenzmatrix (a_{ij}) wie folgt: die Zahl a_{ij} bezeichnet das „Gewicht“ der Kante (i, j), es ist $a_{ij} = 0$, wenn $(i, j) \notin K$ ist; die Adjazenzmatrix ist also genau dann symmetrisch, wenn der Graph „ungerichtet“ ist. In dem Beispiel ist

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Mit unserer Verallgemeinerung entspricht nun aber *jeder* Matrix ein Graph, ggf. schreibt man an jeder Kante ihr Gewicht an.

Welcher Graph gehört dann zur Summe der Adjazenzmatrizen zweier Graphen?
Welcher Graph gehört zur transponierten Adjazenzmatrix? Was ist mit Matrixprodukten?

Hier ist

$$A^2 = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad A^3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad A^4 = A^3$$

Die Einträge in A^2 entsprechen den Wegen der Länge 2 im ursprünglichen Graphen. Die „Erreichbarkeitsmatrix“ $\sum_{i=0}^{?} A^i$ läßt alle möglichen Verbindungen erkennen.

Einem Verkehrsnetz K können wir eine Matrix mit den Einträgen

$$a_{rs} = \text{Entfernung } r \to s, \text{ ohne einen anderen Ort zu durchlaufen,}$$

zuordnen (die Matrix muß angesichts von Einbahnstraßen nicht symmetrisch sein). Wir führen hier eine neue Matrixmultiplikation ein (genauer: nur eine Potenzierung):

$$A^{(2)}_{rs} = \min_{1 \le k \le n} (a_{rk} + a_{ks}).$$

Dann gibt es ein i mit $A^{(i+1)} = A^{(i)}$ und aus dieser Matrix kann man die kürzeste Verbindung zweier Orte ablesen.

Beispiel:

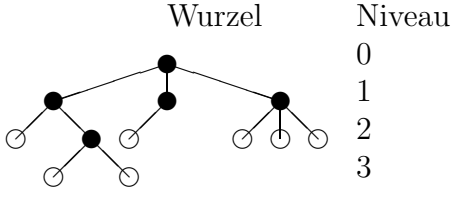
$$A = \begin{pmatrix} 0 & 14 & 8 & 7 & 5 \\ 14 & 0 & 10 & \infty & \infty \\ 8 & 10 & 0 & 3 & \infty \\ 7 & \infty & 3 & 0 & \infty \\ 5 & \infty & \infty & \infty & 0 \end{pmatrix}, \qquad A^{(2)} = \begin{pmatrix} 0 & 14 & 8 & 7 & 5 \\ 14 & 0 & 10 & 13 & 19 \\ 8 & 10 & 0 & 3 & 13 \\ 7 & 13 & 3 & 0 & 12 \\ 5 & 19 & 13 & 12 & 0 \end{pmatrix}.$$

12.1 Bäume

Definition: Ein Graph $G = (E, K)$ heißt Baum, wenn die folgenden äquivalenten Bedingungen erfüllt sind:

1. Je zwei Ecken sind durch genau einen Weg verbunden.
2. G ist zusammenhängend und $|K| = |E| - 1$.
3. G enthält keinen Kreis, aber beim Hinzufügen einer Kante entsteht ein Kreis.

Wir zeichnen eine Ecke eines Baums aus und nennen sie Wurzel. Wenn wir uns einen Graphen als aus Stäben hergestellt vorstellen, die an den Eckpunkten Gelenke haben, so fassen wir einen Baum bei seiner Wurzel und heben ihn an. Dann sieht es etwa so aus:



Man erhält so eine hierarchische Struktur. Die Ecken vom Grad 1 nennt man die Blätter des Baums.

Man nennt einen binären Baum regulär, wenn jeder Eckpunkt, der kein Blatt ist, genau zwei Söhne hat. Also: Jede Ecke hat den Grad 1, 2 oder 3 und es gibt genau einen Knoten vom Grad 2, die Wurzel, $\mid K \mid = \mid E \mid -1$.

Reguläre binäre Bäume haben folgende Eingenschaften:

1. Die Anzahl der Ecken ist ungerade, denn es gibt genau ein $x \in E$ mit $deg(x) = 2$, die anderen Ecken haben einen ungeraden Grad und die Summe der Grade ist eine gerade Zahl.

2. Wenn $|E| = n$ ist, so gibt es $(n+1)/2$ Blätter.

   Beweis: Sei p die Zahl der Blätter, dann gibt es $n-p-1$ Ecken vom Grad 3, also

$$|K| = n - 1 = \frac{1}{2}(p + 3(n-p-1) + 2)$$

$$2n - 2 = 3n - 2p - 1$$

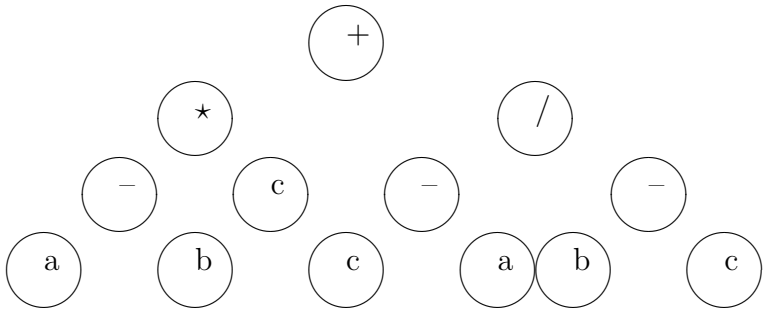
$$-n - 1 = -2p$$

3. Die maximale Zahl der Ecken auf dem Niveau k ist 2^k.

4. Wenn ein Baum die Höhe h und n Knoten hat, so ist $h \geq \log_2(n+1) - 1$.

Durch Bäume lassen sich arithmetische Ausdrücke beschreiben: Die inneren Knoten markieren wir mit $+, -, \star, /$ und die Blätter mit Variablen oder Konstanten.

Beispiel: $(a-b) \star c + (c-a)/(b-c)$



Oft ist es nötig, alle Eckpunkte eines Baums einmal zu durchlaufen. Hierfür gibt es drei rekursive Methoden:

- Inorder-Durchlauf:

  1. Durchlaufe den linken Teilbaum der Wurzel nach Inorder.
  2. Besuche die Wurzel.
  3. Durchlaufe den rechten Teilbaum nach Inorder.

- Preorder-Durchlauf:
  1. Besuche die Wurzel.
  2. Durchlaufe den linken Teilbaum nach Preorder.
  3. Durchlaufe den rechten Teilraum nach Preorder.
- Postorder-Durchlauf:
  1. Durchlaufe den linken Teilbaum nach Postorder.
  2. Durchlaufe den rechten Teilbaum nach Postorder.
  3. Besuche die Wurzel.

Wir illustrieren diese drei Methoden am obigen Beispiel und schreiben die Markierung der besuchten Punkte nacheinander auf:

- Inorder: `a - b * c + c - a / b - c`, dies ist ist der oben angegebene Term, allerdings ohne Klammern.
- Predorder: `+ * - a b c / - c a - c b`, man nennt dies die Präfixnotation oder polnische Notation.
- Postorder: `a b - c * c a - b c - / +`, dies heißt Postfixnotation oder umgekehrte polnische Notation (UPN).

Wir sehen, daß sich bei den beiden letzten Notationen der Term $(a - b) \star \ldots$ ohne Klammern eindeutig darstellen läßt.

Die drei genannten Durchmusterungsalgorithmen sind rekursiver Natur. Wir geben für den Inorder-Durchlauf eines binären Baums einen sequentiellen Algorithmus an. Dabei stellen wir uns den Baum als verkettete Liste mit den Einträgen RECHTS, LINKS (Zeiger auf die beiden Söhne) und INHALT (z.B. ein Name) vor. Weiter haben wir eine globale Variable STACK, einen Stack, den wir mit einer Prozedur PUSH füllen und mit POP leeren. TOP(STACK) ist der oberste Eintrag. EMPTY stellt fest, ob der Stack leer ist und der Zeigerwert NULL gibt an, daß die Liste zu Ende ist.

```
      subroutine inorder(q)
      integer q, p
      p = q
1:    while p<>0
        push(p,stack)
        p = links(p)
      end while
      if not empty(stack)
        p = top(stack)
        pop(stack)
        print *, inhalt(p)
        p = rechts(p)
      end if
      if not (empty(stack) and (p = null))
        goto 1
      end
```

Jeder Knoten wird einmal auf den Stack gelegt und wieder entfernt.

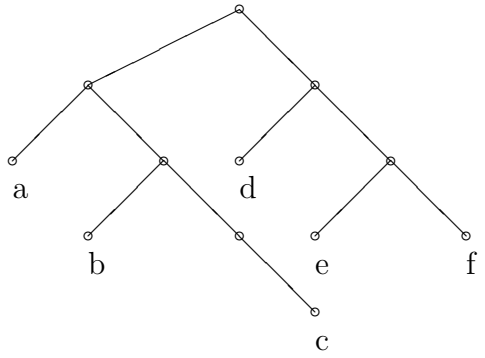
Bäume eignen sich gut zur Strukturierung einer Datenmange, in der immer wieder gesucht werden muß. Seien also die Elemente $x_1, \ldots, x_n$ in die Knotes eines binären Suchbaums eingetragen und zwar so, daß im linken Unterbaum eines Knotens x_i alle kleineren und im rechten Unterbaum alle größeren Elemente zu finden sind. Dann ist

die Zahl der notwendigen Vergleiche, um ein Element zu finden, zu löschen oder an der richtigen Stelle einzufügen, durch die Tiefe des Baums beschränkt. Die besten Resultate wird man erhalten, wenn es gelingt, den Baum möglichst gleichmäßig zu gestalten und nicht zu einem einzigen Zweig ausarten zu lassen.
Beim Abarbeiten eines Suchbaums mit dem Inorder-Verfahren werden die Elemente in der gegebenen Ordnung durchsucht. Man kann sie sich also in der sortierten Reihenfolge ausgeben lassen.

hier: balancierte Bäume

Präfix-Codes

Wir betrachten einen binären Baum, wo jede Kante zu einem linken Sohn mit 0 und jede Kante zu einem rechten Sohn mit 1 markiert ist. Den Blättern ordnen wir Buchstaben eines Alphabets zu, den inneren Knoten wird nichts zugeordnet.



Den zu den Blättern führenden Wege entsprechen die Worte 00, 010, 0111, 10, 110, 111. Keines dieser Worte ist Anfangsstück eines anderen, denn den Anfangsstücken eines Wortes entsprechen innere Knoten und keine Blätter.
Wenn man mit einem solchen Baum ein Alphabet kodiert, so kann man aus gesendeten 0-1-Folgen die einzelnen Code-Werte eindeutig entnehmen. Z.B. bedeutet 00101110111 die Folge *adfc*.
Ermitteln Sie die Baumdarstellung des Präfixcodes aus den Worten 010, 011, 00111, 1010, 111001, 111010, 111011.

Durchmusterung der Kanten eines Graphen

Definition: Ein Baum, der ein spannender Teilgraph eines Graphen G ist, heißt spannender Baum; die restlichen Kanten heißen Sehnen.

Lemma 12.2 *Ein Graph besitzt genau dann einen spannenden Baum, wenn er zusammenhängend ist.*

Beweis: Aus der Existenz eines spannenden Baums folgt der Zusammenhang. Umgekehrt: Ein zusammenhängender Graph ist entweder ein Baum oder er enthält einen Kreis. Wenn wir eine Kreis-Kante weglassen, bleibt der Rest-Graph zusammenhängend. Der Rest-Graph ist einweder ein Baum oder er enthält einen Kreis. Und so weiter.

Wir betrachten zwei Methoden:

1. Breitensuche (breadth-first-search)

   Wir befinden uns in einem Knoten. Als erstes durchlaufen wir alle angrenzenden Kanten. Dann gehen wir zu einem Nachbarknoten und beginnen von vorn.

2. Tiefensuche (depth-first-search)

   Wir befinden uns in einem Knoten. Wir betrachten eine noch nicht begangene Kante und gehen zu deren anderen Endknoten und beginnen von vorn. Wenn es nicht weitergeht, müssen wir umkehren (backtracking).

Wir betrachten die Tiefensuche genauer. Wir werden sehen, daß wir dadurch einen spannenden Baum und eine Nummerierung der Knoten konstruieren können.

Algorithmus zur Tiefensuche:

x_0 sei der Startknoten und $N(x)$ die Nummer von x. Es entsteht der spannende Baum B und die Menge der Rückkehrkanten R besteht aus den übrigbleibenden Sehnen.

1. $x := x_0, i := 0, B := \emptyset, R := \emptyset, N(x) := 0$ für alle x.

2. $i := i + 1, N(x) := i$

3. Wenn es eine noch nicht durchlaufene Kante (x, y) gibt, so gehe nach y; wenn nicht, so gehe zu 5.

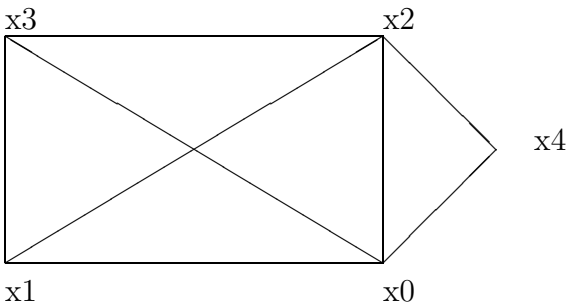
4. Wenn $N(y) = 0$ ist, so wurde y noch nicht besucht. Füge dann (x, y) zu B hinzu, setze $x = y$ und gehe zu 2.

   Wenn $0 \neq N(y) < N(x)$, so war man schon in y. Füge dann (x, y) zu R hinzu und gehe zu 3. (Nun sind wir wieder in x).

5. Wenn es eine Kante $(z, x) \in R$ gibt, so gehen wir nach z zurück. Wir setzen $x = z$ und gehen zu 2.

   Andernfalls ist $x = x_0$ und wir haben alle Kanten durchlaufen.

Wir führen das an einem Beispiel durch:



Wir können z.B. folgenden Durchlauf machen:

$$(x_0, x_1), (x_1, x_2), (x_2, x_0), (x_2, x_3), (x_3, x_0), (x_3, x_1), (x_2, x_4), (x_4, x_0)$$

Dabei entsteht der spannende Baum $B = \{(x_0, x_1), (x_1, x_2), (x_2, x_3), (x_2, x_4)\}$ und als Sehnen verbleiben $R = \{(x_2.x_0), (x_3, x_0), (x_3, x_1), (x_4, x_0)\}$.

Die Tiefensuche hat folgende Eigenschaften.

1. Die Knoten werden mit $1, \ldots, n$ numeriert und die Kanten erhalten eine Richtung, sie werden orientiert.
2. Die Kanten im Baum bilden einen gerichteten spannenden Baum mit der Wurzel x_0; wenn eine Kante (x, y) zum Baum gehört, so gilt $N(x) < N(y)$.
3. Die Kanten in R sind die Sehnen, aus $(x, y) \in R$ folgt $N(x) > N(y)$.

12.2 Gefädelte binäre Bäume

Um einen binären Baum darzustellen, merken wir uns zu jedem Knoten die Adressen `LINKS, RECHTS` seines linken und rechten Sohns. Wenn ein Sohn fehlt, wird diese Adresse auf 0 gesetzt. Bei einem binären Baum mit n Knoten wird also Platz für $n+1$ Nullen gebraucht.
Diesen Speicherplatz kann man sinnvoller nutzen, wenn man sich merkt, daß ein Sohn fehlt (das braucht ein Bit) und sich lieber die Adressen des Vorgängers bzw. Nachfolgers des aktuellen Knotens (bei einer fixierten Durchlaufordnung) merkt. Solche Verbindungen eines Knotens zu seinem Vorgänger/Nachfolger heißen Fäden. Man braucht also in jedem Knoten `p` eine 2-Bit-Information `L(p), R(p)` darüber, ob unter `LINKS` und `RECHTS` die Adressen von Sohnen oder aber Fäden zu finden sind (es liegt nahe, hierfür das Vorzeichen-Bit zu verwenden).

	true	false
`L(p)`	`LINKS` = Sohn	`LINKS` = Vorgänger
`R(p)`	`RECHTS` = Sohn	`RECHTS` = Nachfolger

Der folgende einfache Algorithmus gibt den Inorder-Nachfolger `q = nach(p)` eines Knotens `p` an:

1. Wenn `R(p) = false`, dann `q = RECHTS(p)`,
2. Wenn `R(p) = true` ist, so ersetze `q` durch `LINKS(q)`, solange `L(q) = true` ist.
3. `q` ist der Inorder-Nachfolger von `p`.

Aufgabe: Erstellen Sie einen Algorithmus zur Bestimmung des Vorgängers von `p`.

Im folgenden Bild ist ein binärer Baum nebst Fäden zu den Inorder-Vorgängern bzw. -Nachfolgern dargestellt:

	1	2	3	4	5	6	7	8	9
links	2	4	5	0	3	5	8	8	7
rechts	3	1	7	2	6	3	9	7	0
L	t	t	t	f	f	f	t	f	f
R	t	f	t	f	t	f	t	f	f

Wir bemerken, daß zum Durchlauf eines gefädelten Baums kein Stack verwaltet werden muß.

Der folgende Algorithmus fügt einen Knoten `q` als rechten Sohn des Knotens `p` ein, falls `p` keinen rechten Sohn hatte; wenn der rechte Sohn vorhanden ist, wird `q` zwischen `p` und `rechts(p)` eingefügt.

1. (Parameterübergabe) `rechts(q) = rechts(p), R(q) = R(p), rechts(p) = q, R(p) = true, L(q) = false, links(q) = p.`
2. (War `rechts(p)` ein Faden?) Wenn `R(q) = true`, so setze `links(nach(q)) = q`. Dabei bestimmt man `nach(q)` mittels des obigen Algorithmus.

(Wenn `links` und `rechts` vertauscht werden, so ist `nach` durch `vor` zu ersetzen.)

Wir befassen uns num mit einer Darstellung gefädelter binärer Bäume im Computer. Die Knoten mögen folgende Form haben:

L	links	Typ
R	rechts	Inhalt

Es ist notwendig, alle Algorithmen so zu entwerfen, daß sie auch bei leeren binären Bäumen korrekt arbeiten.
Wenn `t` die Adresse eines Baums ist, so sollte unter der Adresse `nach(t)` der Inhalt des ersten Knotens stehen. Dazu fügen wir einen Kopf `head` ein, dies ist der folgende Knoten:

Sohn	Anfang	
Sohn	head	

Wenn der Baum aber leer ist, setzen wir

Faden	head	
Sohn	head	

Der Baum wächst am linken Sohn des Kopfes. Die Fäden, die auf Null zeigen, werden auf den Kopf gelenkt. In einer Feld-Darstellung eines Baums hätte der Kopf die Adresse 0; wir werden aber eine Listen-Darstellung von Bäumen verwenden, um mit mehreren Bäumen gleichzeitig arbeiten zu können und dabei nicht für jeden (möglicherweise leeren) Baum ein großes Feld bereitstellen zu müssen.

Algorithmus zu Bestimmung des Preorder-Nachfolgers `q = pnach(p)` des Knotens `p` aus der Inorder-Fädlung:

1. Wenn `L(p) = Sohn` ist, setze `q = links(p)` und beende.
2. Setze `q = p`. Wenn `R(p) = Sohn`, gehe zu 3. Sonst wiederhole `q = rechts(q)`, bis `R(q) = Sohn`.
3. Setze `q = rechts(q)`.

Das muß man mal probieren.

Kopieren binärer Bäume:

Am Anfang zeigt `head` auf `t`, `u` zeigt auf `u` (leerer Baum); am Ende zeigt `u` auf `t`.

1. `p = head, q = u`, gehe zu 4.
2. (Ist rechts etwas?) Wenn `p` einen nichtleeren rechten Unterbaum hat, so beschaffe einen neuen Knoten `r` und füge `r` rechts von `q` ein.
3. `Info(q) = Info(p)` (*alles* kopieren.)
4. (Ist links etwas?) Wenn `p` einen nichtleeren linken Unterbaum hat, beschaffe `r` und füge `r` links von `q` ein.
5. (Weitersetzen) `p = pnach(p), q = pnach(q)`
6. Wenn `p = head` ist (das ist genau dann der Fall, wenn `q = rechts(u)`, falls `u` einen nichtleeren rechten Unterbaum hat), so beende; sonst gehe zu 2.

Wir werden versuchen, derartige Baumstrukturen zur Formelmanipulation zu verwenden.

12.3 Paarweises Addieren

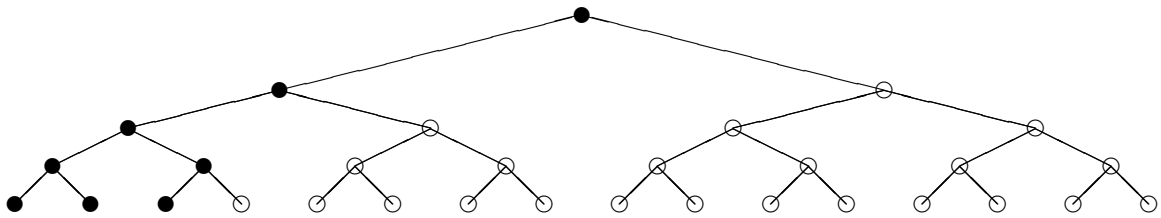
Wir haben dieses Problem schon am Anfang besprochen. Es soll $\sum_{i=1}^{n} = ((a_1+a_2)+(a_3+a_4)+\cdots$ berechnet werden. Dazu stellen wir uns einen binären Baum mit n Blättern vor, in denen wir die a_i eintragen. Dann belegen wir schrittweise jeden Knoten durch die Summe seiner Söhne, die wir daraufhin entfernen. In der Wurzel erhalten wir die Summe.
Das folgende Programm stellt eine einfache Implementation dieses Gedankens dar.

```
static public class paar
{
  int bb = 8000;
  float t[] = new float[bb * 2]
  float s, v;

public static int nextpower(w)
{
  int w, p2 = 1;
  while (w > p2)
    p2= p2 * 2;
  return p2;
}

public static void main(String a[])
{
  int i, k, kk;
  kk= bb;
  k= nextpower(kk);
```

```
  for (i = k, i <= kk+k-1; i++)
  {
     v = Math.random();
     t(i) = v * i;
  }
  for (i = kk+k; i <= 2*k; i++)
    t[i]= 0.0;
  while (k > 1)
  {
    i = 0;
    while (i < k)
    {
      t[(k+i)/2] = t[k+i] + t[k+i+1];
      i = i + 2;
    }
    k= k / 2;
  }
}
```

12.4 Baum-Darstellung des Speichers

Unter dem Gesichtspunkt der Langzahlarithmetik erscheint es als sinnvoll, den Speicher in Blöcken der Größe $2^k,\ 2^{k+1}, \ldots, 2^m$ zu organisieren (bei der Multiplikation verdoppelt sich der Platzbedarf). Zwei benachbarte Blöcke der Größe 2^i können zu einem Block der Größe 2^{i+1} verbunden werden.
Wir stellen uns also den Speicher als binären Baum, also hierarchisch geordnet, vor.



Jeder Vater der Größe m hat zwei Söhne der Größe $m/2$. Wir teilen jedem Knoten mit, ob er frei ist — in diesem Fall sind auch all seine Nachkommen frei, oder ob er besetzt ist — in diesem Fall sind auch all seine Vorgänger besetzt (d.h. nicht im ganzen Ausmaß zu vergeben). Wenn ein Vater besetzt ist, so kann man bei den Söhnen nachfragen, welcher ggf. frei ist. Der Baum möge N Blätter besitzen, er hat also die Tiefe $\log(N)$. Wie wir früher gesehen haben, sind die Adressen der Söhne in einem binären Baum leicht aus der des Vaters zu berechnen (und umgekehrt), wenn der Baum in einem linearen Feld dargestellt wird.

Es sind also folgende Operationen nötig:
Wenn ein freier Block der Größe s gesucht wird, so durchläuft man im entsprechenden Niveau des Baums alle Knoten, bis man einen freien gefunden hat. Die Blöcke der

Größe s haben die Tiefe $t = ld(N) - ld(s) - 1$, sie haben die Nummern $2^t, \ldots, 2^{t+1} - 1$. Wenn ein freier Knoten i gefunden wurde, so sind all seine Vorfahren i div 2, i div 4, ... als belegt zu melden, und die Knoten des Unterbaums mit der Wurzel i ebenfalls. Den Unterbaum durchlaufen wir mit einer der behandelten Durchlaufungsmethode.
Wenn ein Knoten freigegeben wird, sind die Nachkommen freizugeben. Wenn der Bruder auch frei ist, so ist der Vater freizugeben und mit diesem ist ebenso zu verfahren.

Aus der Nummer eines gefundenen freien Knotens berechnet man die Stelle in `memo`, wo man seine Daten eintragen kann.

Es folgt eine Implementation:

```
import java.io.*;

public class baum
{
public static int MaxLevel = 7, N = 128, LU = N / 2, Width = N * 2;

// N = 2 ^ maxlevel, LU = N / 2 (= linke untere Ecke
public static boolean tree[] = new boolean[N + 1];

public static int ld(int i) //             { Log zur Basis 2 }
{
  int l, j;
  l = -1;
  j = 1;
  while (j < i)
  {
    j = j * 2;
    l++;
  }
  return l + 1;
}

public static int two2(int i)
{
  int j, t = 1;
  for (j = 1; j <= i; j++)
    t = 2 * t;
  return t;
}

public static int left(int i)
{
  if (i < LU)
    return 2*i;
  else
```

```
    return 0;
}

public static int right(int i)
{
  if (i < LU)
    return 2*i + 1;
  else
    return 0;
}

public static int father(int i)
{
  if (i > 0)
    return i/2;
  else
    return 0;
}

public static int brother(int i)
{
  if (2*(i/2) != i)   // odd
    return i - 1;
  else
    return i + 1;
}

public static int search4(int s) //  { sucht einen Platz der groesse s }
{
  int z, i;
  z = two2(ld(N) - ld(s) - 1);
  for (i = z; i <= 2*z - 1; i++)
    if (tree[i])
      return i;
  if (!tree[2*z - 1])
    return 0;
  return 0;
}

public static void putinorder(int i, boolean b)
{
  int[] stack = new int[LU];
  int j = i, st = 0;
  while (true)
  {
```

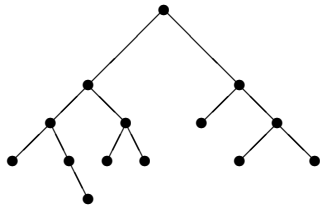
```
    while (j != 0)
    {
      st++;
      stack[st] = j;
      j = left(j);
    }
    if (st > 0)
    {
      j = stack[st];
      st--;
      tree[j] = b;
      j = right(j);
    }
    if ((st <= 0) && (j > 0))
      break;
  }
}

public static void get(int i)
{
  int j = i;
  while (tree[j])
  {
    tree[j] = false;
    j = j / 2;
  }
  putinorder(i, false);
}

public static void free(int i)                    //gibt Stelle i frei
{
  boolean weiter = true;
  while (weiter)
  {
    if (i == 1)
      break;
    tree[i] = true;
    putinorder(i, true);
    System.out.print(" freigegeben: " + i);
    if (tree[brother(i)])         //unterhalb von i kann nichts frei sein }
      i = father(i);
    else
      weiter = false;
  }
}
```

```
static char bild[][] = new char[MaxLevel+2][Width+2];

public static void drawtree(int Level)
{
  int x, i, j, C;
  if (Level < MaxLevel)
    drawtree(Level + 1);
  x = Width / two2(Level);
  C = Width / two2(Level - 1);
  j = two2(Level - 1);
  for (i = 0; i < j; i++)
  {
    if (tree[j + i])
    {
      bild[Level][x] = '<';
      bild[Level][x+1] = '>';
    }
    else
    {
      bild[Level][x] = (char)((i+j) / 10 + 48);
      bild[Level][x+1] = (char)((i+j) % 10 + 48);
    }
    x = x + C;
  }
}

public static void main(String arg[])
{
  int i;
  for (i = 1; i <= N; i++)
    tree[i] = true;
  System.out.println("belegte Bloecke");
  drawtree(1);
  for (i = 0; i < MaxLevel; i++)
    System.out.println(bild[i]);
  while (true)
  {
    System.out.print("(zum Freigeben: 0); angeforderte Blockgroesse: ");
    i = B.readint();
    if (i <= 0)
      break;
    i = search4(i);
    if (i == 0)
    {
```

```
      System.out.println("kein Platz");
      break;
    }
    get(i);
    System.out.println("Nummer des belegten Blocks: " + i);
    System.out.println("belegte Bloecke");
    drawtree(1);
    for (i = 0; i < MaxLevel; i++)
      System.out.println(bild[i]);
    System.out.println();
  }
  while (true)
  {
    System.out.print("(zum Belegen: 0); streiche Block mit Nummer: ");
    i = B.readint();
    if (i <= 0)
      break;
    free(i);
    System.out.println("belegte Bloecke");
    drawtree(1);
    for (i = 0; i < MaxLevel; i++)
      System.out.println(bild[i]);
    System.out.println();
  }
}
}
```

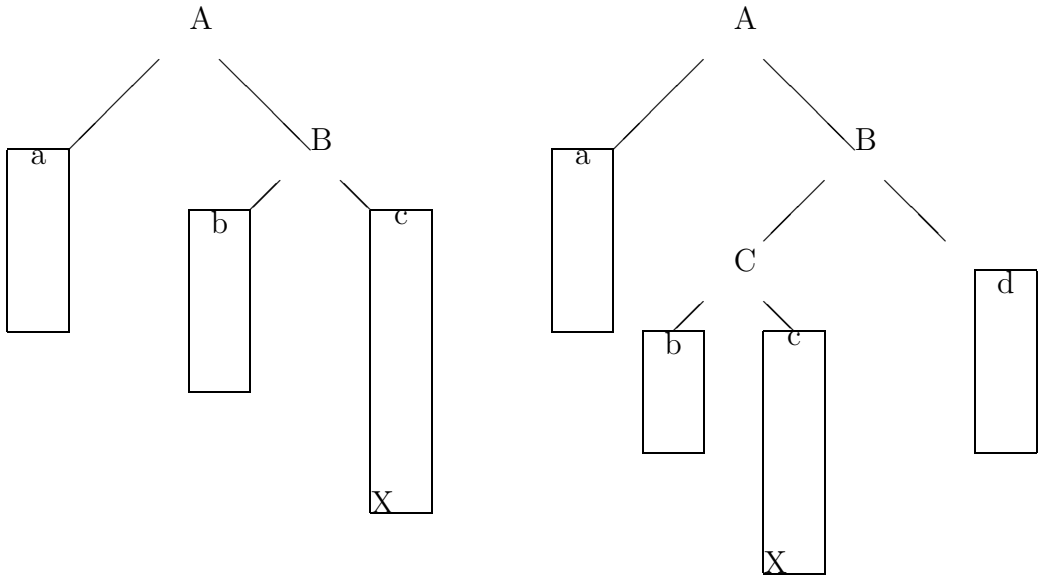
12.5 Balancierte Bäume

Ein binärer Baum heißt balanciert, wenn für jeden Knoten gilt: Die Höhne des rechten und des linken Unterbaums unterschieden sich höchstens um 1.



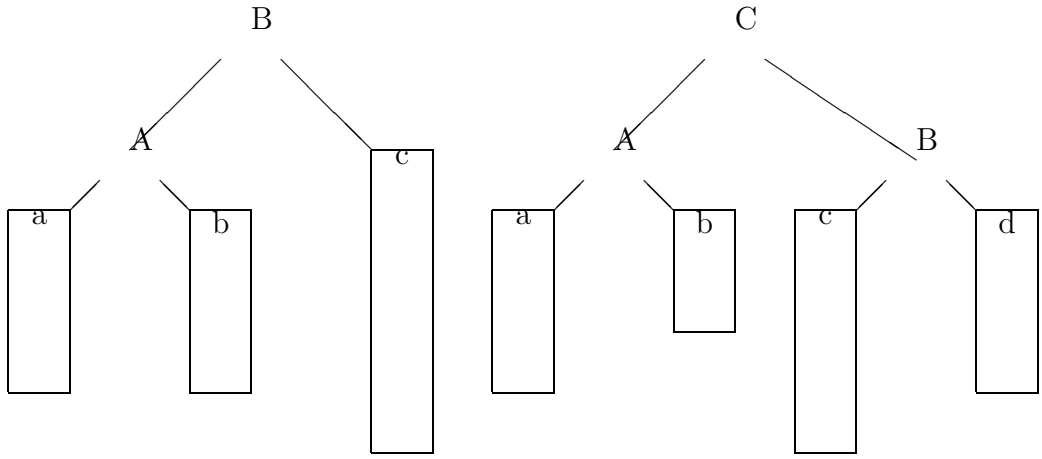
Zum Beispiel:

Es ist klar, daß die Suche in balancierten Bäumen effektiver als in unbalancierten durchgeführt werden kann.



Beim Einfügen eines Knotens (bei X) kann die Balance verlorengehen. Dabei sind zwei Fälle möglich (Die durch Kästen dargestellten Unterbäume haben die Höhe h, $h-1$, $h+1$):

Durch „Rotieren“ kann sie wiederhergestellt werden.



13 Computeralgebra

13.1 Langzahlarithmetik: Rohe Gewalt

Zu Beginn will ich eine sehr einfache Implementierung für eine Langzahlarithmetik angeben, die zwar sehr durchsichtig, aber nicht sehr effizient ist.

```
      program big10

      integer a(60), b(60), c(60), zehn, w

      common zehn
1     format ('+', A)
      zehn = 10
5     print *, ' '
      write (*, 1) 'a :'
      call eingabe(a)
      call ausgabe(a)
      write (*, 1) 'b :'
      call eingabe(b)
      write (*, 1) 'a + b :'
      call add(a, b, c)
      call ausgabe(c)
      write (*, 1) 'a - b :'
      call sub(a, b, c)
      call ausgabe(c)
      call mult(a, b, c)
      write (*, 1) 'a * b :'
      call ausgabe(c)
      w = 10
      call power(a, w, c)
      call ausgabe(c)
      goto 5
      end

      subroutine kontr (a)
      integer a(60), i
30      format (60I2)
        write (*, 30) (a(i), i = 1, 60)
        read (*, '(A)') c
      end

      subroutine init(zahl)
      integer zahl(60)
      integer i
      do i= 1 , 60
        zahl(i)= 0
      end do
      end

      subroutine put9(zahl)
      integer zahl(60), i, zehn
      common zehn
      do i= 1 , 60
        zahl(i)= zehn-1
      end do
      end

      subroutine ausgabe (zahl)
      integer zahl(60)
      character*60 s
      character blank
      parameter (blank = '.')
      integer i
      logical fuehrendenull
      fuehrendenull= .true.
      print *, ' '
10    format ('+', A)
20    format ('+', I1)
      do i= 1 , 60
        if (fuehrendenull .and.
     +  (zahl(i) .eq. 0) .and.
     +  (i .lt. 60))
     +  then
          s(i:i) = blank
        else
          s(i:i) = char(zahl(i) + ichar('0'))
          fuehrendenull= .false.
        end if
      end do
      print *, s
      print *, ' '
      end
```

```
subroutine malzehn(zahl1, zahl2)
integer zahl1(60), zahl2(60)
integer i
if (Zahl1(1) .ne. 0) then
  print *, 'Ueberlauf * 10'
  stop
else
  do i= 2 , 60
    zahl2(i-1)= zahl1(i)
  end do
  zahl2(60)= 0
end if
end

subroutine setze(zahl, wert)
integer zahl(60), wert, i, zehn
common zehn
call init(zahl)
i= 60
do while (wert .gt. 0)
  zahl(i)= mod(wert, zehn)
  wert= wert / zehn
  i = i - 1
end do
end

subroutine add(s1, s2, summe)
integer s1(60), s2(60), summe(60)
integer i, carry, zehn
common zehn
carry= 0
i= 60
do while (i .gt. 0)
  carry= s1(i) + s2(i) + carry / zehn
  summe(i)= mod(carry , zehn)
  i = i - 1
end do
if (carry .gt. zehn-1) then
  print *, 'Ueberlauf +'
  stop
end if
end
```

```
subroutine sub(s1, s2, diff)
integer s1(60), s2(60), diff(60)
integer i, carry, zehn
common zehn
carry= 0
do i= 60, 1, -1
  carry= s1(i) - s2(i) + carry
  if (carry .lt. 0) then
    diff(i)= carry + zehn
    carry= -1
  else
    diff(i)= carry
    carry= 0
  end if
end do
end

integer function anf(a)
integer a(60)
integer i
i= 1
do while ((i .le. 60) .and. (a(i) .eq. 0))
  i = i + 1
end do
anf= i
end

subroutine mult (f1, f2, produkt)
integer f1(60), f2(60), produkt(60), i, anf
integer prod(60), f2no(60)
f2no= f2
call init(prod)
i= anf(f2no)
do while (i .le. 60)
  call malzehn(prod, prod)
  do while (f2no(i) .gt. 0)
    call add(f1, prod, prod)
    f2no(i) = f2no(i) - 1
  end do
  i = i + 1
end do
produkt= prod
end
```

```
      integer function length(s)
      character*60 s
      integer i
      i = 1
      do while ((s(i:i) .ge. '0')
     +     .and. (s(i:i) .le. '9'))
        i = i + 1
      end do
      length = i - 1
      end

      subroutine eingabe(a)
      integer a(60), length
      character*60 s
      integer h(60), g(60)
      integer i
      read (*, '(A60)') s
      call init(a)
      do i= 1 , length(s)
        call setze(h, ichar(s(i:i)) - ichar('0'))
        call malzehn(a, g)
        call add(g, h, a)
      end do
      end
      subroutine power(a, hoch, a2)
      integer a(60), a2(60), acopy(60)
      integer hoch
      acopy= a
      call setze(a2, 1)
      do while (hoch .gt. 0)
        call mult(acopy, a2, a2)
        hoch = hoch - 1
      end do
      end
```

13.2 Wirkliche Langzahlarithmetik

Im diesem Abschnitt werden wir sehen, wie (innerhalb der durch den Rechner gesetzten Speichergrenzen) eine exakte Arithmetik zu implementieren ist. Als Hilfsmittel dazu ist eine verfeinertere Arithmetik für lange ganze Zahlen nötig, mit der wir uns hier befassen.
Wir beginnen nicht mit natürlichen Zahlen, sondern wollen uns auch das Vorzeichen einer Zahl merken.
Wir stellen eine ganze Zahl n in einem Stellensystem dar, wie dies seit der Einführung der arabischen Zahlen üblich ist. Dazu wählen wir eine Basiszahl $B > 1$ und können die Zahl n in eindeutiger Weise als

$$\sum_{n=0}^{L} n_i B^i,\ 0 \leq n_i < B,$$

darstellen, die Werte $n_0, n_1, \ldots, n_L$ nennen wir die Ziffern der Zahl n, die Zahl L nennen wir die Länge von n.

Wenn wir $B \leq 256$ wählen, passen die n_i jeweils in ein Byte, bei $B \leq 65536 = 2^{16}$ passen sie in ein Maschinenwort eines PC.
Um alle Informationen über unsere langen Zahlen anzuspeichern, gibt es mehrere Möglichkeiten:

1. Wir legen eine maximale Zahllänge fest:

```
NumSize = 64
```

und legen die Ziffern einer langen Zahl in einem Feld ab:

```
type Digit = byte;
IntArr = array [1..NumSize] of Digit;
IntNumber = record
l: integer;     { Länge }
m: IntArr;     { Ziffernfolge }
n: boolean;     { negativ ? }
end;
```

dies ist die einfachste Methode. Sie hat zwei Nachteile: Bei kurzen Zahlen (l < NumSize) wird Speicherplatz verschenkt, andererseits addieren sich bei Multiplikationen oft die Längen der Faktoren, die vorgegebene Maximalgrenze ist schnell erreicht.

2. Wir legen nur die notwendigen L Ziffern in einer verketteten Liste ab:

```
type IntList = POINTER TO NatNumRec;
IntNumRec = record
  m: Digit;     { Ziffer }
  n: boolean;     { negativ ? }
  f: IntList; { Adresse der nächsten Ziffer }
end;
```

So können wir „wirklich“ beliebig lange Zahlen anspeichern, üblicherweise weist man der letzten gültigen Ziffer im Feld `f` eine leicht zu erkennende Adresse zu, etwa `NIL` (= 0). Wenn man eine neue Zahlvariable belegen will, muß man sich zuerst Speicherplatz für diese Variable holen, dafür gibt es Allockierungsfunktionen (z.B. NEW). Für eine Zahl der Länge L benötigen wir L * (SizeOf(byte) + SizeOf(ADDRESS)), auf einem PC also $5L$ Byte. Diese Möglichkeit ist (wegen der gefallenen Schranke) besser als die erste, aber sehr speicherintensiv.
Warum haben wir eigentlich nur ein Byte für eine Ziffer verwendet, man könnte auch ein Wort (= CARDINAL) oder einen noch größeren Speicherbereich (LongWord, LONGCARD) verwenden. Nun, die Rechenoperationen mit langen Zahlen werden später auf Rechenoperationen mit Ziffern zurückgeführt werden, und auf der Maschinenebene gibt es elementare Befehle, die diese Operationen mit Bytes ausführen. Das Produkt von zwei Bytes paßt in ein Doppelbyte, dies läßt sich leicht in zwei Bytes zerlegen. Wir können also als Zifferntyp den größten Typ von Maschinenzahlen wählen, für den es einen doppeltsogroßen Maschinenzahltyp gibt.
Wenn wir also neu

```
type Digit = CARDINAL;
```

festlegen, so sinkt der Adressierungsaufwand.

3. Als Kompromiß zwischen beiden Varianten bietet es sich an, die Ziffern in einem dynamischen Feld zu speichern, dies ist zwar echt begrenzt, aber die Grenze ist doch recht hoch:
Wir beginnen von vorn:

```
const NumSize = 32000;
type Digit = CARDINAL;
IntArr = array[1..NumSize] of Digit;
IntRec = record
  l: integer;     { Länge }
  n: boolean;     { negativ ? }
  m: IntArr;     { Ziffernfolge }
end;
IntNumber = POINTER TO IntRec;
```

Beachten Sie bitte, daß im Gegensatz zur allerersten Methode das Record-Feld `m`, dessen Größe variabel ist, als letztes in der Definition des Datentyps aufgeführt ist. Der Speicherplatz für solch ein Record wird in dieser Reihenfolge angelegt, wenn `n` hinter `m` käme, würde diese Eintragung an einer Stelle erfolgen, für die gar kein Speicherplatz angefordert wird.
Bevor wir eine Variable N vom Typ IntNumber belegen können, müssen wir hierfür Speicherplatz holen. Hier ist die Verwendung von NEW unangemessen, denn wir wollen ja für eine Zahl der Länge L nur L * SizeOf(Digit)+SizeOf(integer)+SizeOf(boolean) Bytes belegen. Also verwenden wir

```
ALLOCATE(N, L· SizeOf(Digit)+SizeOf(integer)+SizeOf(boolean));
```

Die maximale auf diese Weise darstellbare Zahl hat etwa 20000 Dezimalstellen (etwa 10 A4-Druckseiten), das dürfte (für's erste) reichen.
Wenn wir aus zwei Zahlen X, Y eine neue Zahl Z berechnen wollen, müssen wir den Platzbedarf von Z kennen. Hierfür hat man folgende Abschätzungen:
Falls $Z = X + Y$ oder $Z = X - Y$, so ist die Länge L_Z von Z kleiner oder gleich $\mathrm{Max}(L_X, L_Y) + 1$. Falls $Z = X \cdot Y$ und $X, Y \neq 0$ sind, so ist $L_Z \leq L_X + L_Y$.
Bevor wir mit den arithmetischen Operationen beginnen, wollen wir überlegen, wie die Basis B unseres Zahlsystems zweckmäßigerweise gewählt werden sollte.
Wenn wir für B eine Zehnerpotenz wählen, so sind die Ein- und Ausgabeoperationen sehr leicht zu implementieren. Leider sind diese Operationen diejenigen, die am allerseltensten aufgerufen werden, meist nur am Beginn und am Ende eines Programmlaufs. Wenn wir $B = 32768$ setzen, so bleibt das höchste Bit jeder Ziffer frei, wie im Fall von $B = 10^k$ wird so Speicher verschenkt. Wir haben oben die Festlegung

```
Digit = byte
```

durch

```
Digit = CARDINAL
```

ersetzt. Der Grund für diese Änderung (Adressierungsaufwand) ist inzwischen entfallen, wir verwenden ja dynamische Felder. Jenachdem, für welche Ziffer-Größe man sich entscheidet, wäre B = 256 oder B = 65536 zu wählen. Die internen Darstellungen einer Zahl auf diese oder die andere Weise unterscheiden sich um kein Bit, nur die

Länge L ist im ersten Fall doppelt so groß wie im zweiten. Wenn im folgenden also Laufanweisungen der Länge L (oder gar der Länge L^2) auftreten, ist eine kürzere Darstellungslänge eventuell von Vorteil. Da im Fall B = 65536 ständig CARDINALs auf LONGCARDs „gecastet“ und LONGCARDs in CARDINALs zerlegt werden, läßt sich die richtige Wahl nicht rechner- und compilerunabhängig treffen. Man erlebt die größten Überraschungen.

Wir beginnen nun mit der Beschreibung von Implementationen der Rechenoperationen. Wir definieren zunächst

```
CONST basis = LONGCARD(65536);
```

Die einfachste Operation ist die Addition von Zahlen gleichen Vorzeichens. Die enstprechenden Stellen der Summanden werden als LONGCARDs addiert, der Rest `w mod basis` ist die entsprechende Stelle der Summe, der Quotient `w basis` ist der Übertrag, er ist gleich 0 oder 1.

```
PROCEDURE al(a, b: IntNumber;
             VAR c: IntNumber);
(* Add longs, signs are equal *)
VAR i, max, min, w: LONGCARD;
    cc: IntNumber;
BEGIN
  IF a^.l > b^.l THEN
    min := b^.l; max := a^.l
  ELSE
    min := a^.l; max := b^.l;
  END;
  newl(c, max + 1);
  { allockiert Platz f"ur c }
  clear(c);
  { belegt c mit Nullen }
  w:= 0;
    FOR i:= 1 TO min DO
      w:= w + a^.m[i] + b^.m[i];
      c^.m[i]:= w MOD basis;
      w:= w DIV basis;
    END;
    { a oder b ist abgearbeitet }
    IF max = l THEN
    { von a noch ein Rest }
      FOR i:= min + 1 TO max DO
        w:= w + a^.m[i];
        c^.m[i]:= w MOD basis;
        w:= w DIV basis;
      END
    ELSE
    { oder von b noch ein Rest }
      FOR i:= min + 1 TO max DO
        w:= w + b^.m[i];
        c^.m[i]:= w MOD basis;
        w:= w DIV basis;
      END;
    END;
  IF w>0 THEN
  { an letzter Stelle "Ubertrag? }
    c^.m[max +1]:= w;
  ELSE
    newl(cc, c^.l-1);
    { sonst c verk"urzen }
    cc^.n:= c^.n;
    FOR i:= 1 TO cc^.l DO
      cc^.m[i]:= c^.m[i];
    END;
    lrWegl(c);
    { altes c wegwerfen }
    c:= cc;
  END;
  c^.n:= a^.n;
END al;
```

Wir kommen nun zur Subtraktion, und zwar dem einfachen Fall, daß die Operanden das gleiche Vorzeichen haben und der Subtrahend den größeren Betrag hat. Die einzelnen Stellen werden als LONGINTs subtrahiert und ein Übertrag von der nächsthöheren Stelle des Subtrahenten subtrahiert.

```
PROCEDURE sl(a, b: IntNumber;
             VAR c: IntNumber);
(* Subtract longs,
signs are equal, a > b *)
VAR i, j, jj: CARDINAL; cc: IntNumber;
    lh, li, ln, lb: LONGINT;
BEGIN
  newl(c, a^.l);
  clear(c);   { c:= 0 }
    ln := a^.m[1];
    FOR i:= 1 TO b^.l DO
    { zun"achst a und b }
      li:= ln ;
      ln := a^.m[i+1];
      lb:=(b^.m[i];
      lh:= li - lb;
      IF lh < 0 THEN
      { "Ubertrag }
        DEC(ln);
        INC(lh, basis);
      END;
      c^.m[i]:= lh;
    END;
    FOR i:= b^.l+1 TO a^.l DO
    { nun Rest von a }
      li:= ln ;
      ln := a^.m[i+1];
      IF li < 0 THEN
        DEC(ln);
        INC(li, basis);
      END;
      c^.m[i]:= li;
    END;
    j:= l;
    WHILE (j>0) AND (c^.m[j]=0) DO
    { wieweit ist c belegt ? }
      DEC(j);
    END;
  IF j=0 THEN
    lrWegl(c);   { c = 0 }
    RETURN;
  END;
  IF j < a^.l THEN
    newl(cc, j);
    { c verk"urzen }
    clear(cc);
    FOR i:= 1 TO j DO
      cc^.m[i]:= c^.m[i];
    END;
    lrWegl(c);
    c:= cc;
  END;
  c^.n:= a^.n;
  { Ergebnis hat Vorzeichen
  der gr"o"seren Zahl }
END sl;
```

Nun kommt der allgemeine Fall der Addition und Subtraktion ganzer Zahlen.

```
PROCEDURE add(a, b:IntNumber;
              VAR c: IntNumber);
(* c:= a + b *)
VAR s: CARDINAL;
BEGIN\p
  IF a^.n = b^.n THEN
  { gleiches Vorzeichen }
    al(a,b,c)
  ELSE\p
    s:= compabs(a, b);
    { Vergleiche Absolutbetr"age }
    CASE s OF
      2: sl(b, a, c);   { b > a }
      |1: sl(a, b, c);  { a > b }
      |0: c:= NIL;      { a = b }
    END;
  END;
END add;
```

```
PROCEDURE sub(a, b:IntNumber;
              VAR c: IntNumber);
(* c:= a - b *)
VAR s: CARDINAL;
BEGIN
  IF a=NIL THEN  { wenn a = 0 }
    lrCopi(b, c);   { c:= b }
    IF c$<>$NIL THEN
      c^.n:= NOT c^.n  { c:= -c }
    END;
    RETURN;
  END;
  IF b=NIL THEN  { wenn b = 0 }
    lrCopi(a, c);  { c:= a }
    RETURN;
  END;
```

```
  IF a^.n <> b^.n THEN
  { verschiedene Vorzeichen ? }
    al(a,b,c)  { dann addieren }
  ELSE
    s:= compabs(a, b);
    { sonst Vergleich }
    CASE s OF
      1: sl(a, b, c);   { a > b }
      |2: sl(b, a, c);
          c^.n:= NOT b^.n;
          { a < b, Vorzeichen ! }
      |0: c:= NIL;   { a = b }
    END;
  END;
END sub;
```

Wir kommen nun zur Multiplikation.
Zunächst führen wir eine Hilfsprozedur an, die ein effektiver Ersatz für folgende Prozeduraufrufe ist:

```
al(a, b, c);
{ hier steckt mindestens eine Allockierung dahinter }
wegl(a); a:= c;
```

Die Variable a wird nur einmal angelegt.

```
PROCEDURE a2((*VAR*) a, b: IntNumber);
(* a:= a + b, signs are equal,
a already exists ! *)
VAR i, max, min : INTEGER;
    w: LONGCARD;
BEGIN
  min:= b^.l; max:= a^.l; w:= 0;
  WITH a^ DO
    FOR i:= 1 TO min DO
      w:= w + a^m[i] + b^.m[i];
      m[i]:= w MOD basis;
      w:= w DIV basis;
    END;
    FOR i:= min +1 TO max  DO
      w:= w + a^.m[i];
      m[i]:= w MOD basis);
      w:= w DIV basis;
    END;
    IF w>0 THEN
      m[max +1]:= w;
    END;
  END;
END a2;
```

Die Multiplikation wird genauso durchgeführt, wie Sie es in der Schule gelernt haben: Ein Faktor wird mit einer einstelligen Zahl multipliziert und das Ergebnis wird verschoben, dies leistet die Unterprozedur `mh`. Diese Werte werden aufaddiert, dies leistet `a2`. Zu beachten ist, daß die Länge der abzuarbeitenden FOR-Schleife für die Rechenzeit eine wesentliche Bedeutung hat; es wird die kürzere Schleife gewählt.
(Die von nun an auftretenden Vorsilbe „lr“ in den Prozedurnamen wird verwendet, um daran zu erinnern, daß es sich um Operationen mit langen rationalen Zahlen handelt.

```
PROCEDURE lrMl(a, b:IntNumber;
               VAR c: IntNumber);
(* mult long, c:= a * b *)
VAR hh: IntNumber;
    la, lb, i, j, hl: CARDINAL;
    h: LONGCARD;
  PROCEDURE mh(a: IntNumber;
               b: LONGCARD;
               s: CARDINAL);
  VAR i: CARDINAL;
  BEGIN
    h:= 0;
    WITH a^ DO
      FOR i:= 1 TO l DO
        h:= h + m[i] * b;
        hh^.m[i+s]:= h MOD basis;
        h:= h DIV basis;
      END;
      hh^.m[l+s+1]:= ORD(h);
    END;
    IF h=0 THEN
      hh^.l:= a^.l+s
    ELSE
      hh^.l:= a^.l+s+1;
    END;
  END mh;
BEGIN (* lrMl*)
  la:= a^.l;  lb:= b^.l;
  hl:= la+lb;
  newl(c, hl); clear(c);
  newl(hh, hl); clear(hh);
  IF la < lb THEN
  { k"urzere Schleife au"sen ! }
    FOR i:= 1 TO la-1 DO
      IF a^.m[i]<>0 THEN
        mh(b, a^.m[i], i-1);
        a2(c, hh);
        FOR j:= i TO hh^.l DO
          hh^.m[j]:= 0;
        END;
      END;
    END;
    mh(b, a^.m[la], la-1);
    a2(c, hh);
  ELSE
    FOR i:= 1 TO lb-1 DO
      IF b^.m[i]<>0 THEN
        mh(a, b^.m[i], i-1);
        a2(c, hh);
        FOR j:= i TO hh^.l DO
          hh^.m[j]:= 0;
        END;
      END;
    END;
    mh(a, b^.m[lb], lb-1);
    a2(c, hh);
  END;
  IF c^.m[hl]=0 THEN
    shrink(c);
  END;
  c^.n:= a^.n <> b^.n;
  hh^.l:= hl;
  lrWegl(hh);
END lrMl;
```

Nun kommen wir zur Division, das war schon in der Schule etwas schwieriger. Gegeben sind zwei Zahlen

$$a = \sum a_i B^i \text{ und } b = \sum b_i B^i,$$

gesucht sind Zahlen q und r mit

$$a = bq + r,\ 0 \le r < b.$$

Wenn $a < b$ sein sollte, setzen wir $q = 0$ und $r = a$ und sind fertig. Andernfalls muß man die einzelnen Stellen von q nun erraten. Wir bezeichnen wieder mit $L(a)$ die Länge der Zahl a. Als Länge von q ist etwa $L(a) - L(b)$ zu erwarten, wir setzen

$$Q = a_{L(a)} \text{ div } b_{L(B)}$$

$$q = Q \cdot B^{L(a)-L(b)}$$

und prüfen, ob

$$0 \le r = a - bq < b$$

ist. Wenn nicht, so haben wir zu klein geraten, wir erhöhen Q, oder r ist negativ, dann haben wir zu groß geraten, wir erniedrigen Q und probieren es nochmal. Wenn r endlich im richtigen Bereich liegt, haben wir die erste Stelle von q gefunden, wir ersetzen nun a durch r und berechnen in analoger Weise die nächste Stelle von q.

Das beschriebene „Falschraten“ kann sehr häufig vorkommen und die Rechenzeit erheblich belasten. Seit Erscheinen der Algorithmenbibel von D. Knuth wird allenthalben vorgeschlagen, die Zahlen a und b durch „Erweitern“ so anzupassen, daß $b \geq B/2$ ist, dadurch ist gesichert, daß der geratene Quotient Q um höchstens 2 vom richtigen Quotienten abweicht. Ich hatte dies erst nachträglich in meine Implementation eingefügt und habe tatsächlich eine Beschleunigung erlebt.
Einen anderen Vorschlag hat W. Pohl (z.Z. Kaiserslautern) gemacht, der deutlich besser ist:
Wir handeln den Fall, daß b eine einstellige Zahl ist, extra ab, das ist auch recht einfach. Nun bilden wir aus den jeweils beiden ersten Stellen von a und b Long-Cardinal-Zahlen und wählen deren Quotienten als Näherung für Q, da liegen wir schon ganz richtig.
Ich will uns den Abdruck der Divisionsprozedur ersparen, die nimmt etwa zwei Druckseiten ein.
Die vorgestellten Prozeduren für die Grundrechenarten sind in Pascal oder MODULA geschrieben. Bei der Multiplikation sehen Sie, daß häufig Wort-Variable auf Longint-Variable „gecastet“ werden, weil das Produkt zweier 16-Bit-Zahlen eben eine 32-Bit-Zahl ist. Es ist vorstellbar, daß man effektiver arbeiten könnte, wenn man die Fähigkeiten des Prozessors besser nutzen würde. Henri Cohen schreibt, daß eine Beschleunigung um den Faktor 8 erreicht werden würde, wenn man in Assemblersprache programmiert. Meine Erfahrungen gehen noch weiter: In einer früheren Version meines CA-Systems, wo die Zahllänge konstant war (64 Byte), hat Sven Suska für die Addition und die Multiplikation Assembler-Routinen geschrieben, allein deren Einsatz brachte eine Beschleunigung um den Faktor 6. Auch für die Division hat er eine Routine geschrieben, die leider den Nachteil hatte, ab und zu den Rechner zum Absturz zu bringen. Wenn sie aber fehlerfrei funktionierte, so erbrachte das nochmals eine Beschleunigung un den Faktor 6. Assembler-Routinen sind allerdings schwer zu transportieren; die erwähnten funktionierten nur im Real-Modus des 286er Prozessors, wo man maximal 500 KByte Speicher zur Verfügung hat. Außerdem waren sie sehr umfangreich (über 18 KByte Quelltext) und für einen Laien wie mich völlig unverständlich.
Cohen schlägt vor, nur einige Grundfunktionen in Assembler zu schreiben. Dazu sollen zwei globale Variable namens `remainder` und `overflow`, die überall bekannt sind, geschaffen werden. Die Zahl-Basis sei `M = 2 ^16`, `a, b` und `c` seien vorzeichenlose 2-Byte-Zahlen.
Die Grundfunktionen sollen folgendes leisten:

```
c = add(a, b)      : a + b = overflow * M + c
c = addx(a, b)     : a + b + overflow = overflow * M + c
c = sub(a, b)      : a - b = c - overflow * M
c = subx(a, b)     : a - b - overflow = c - overflow * M
c = mul(a, b)      : a * b = remainder * M + c
c = div(a, b)      : remainder * M + a = b * c + remainder
c = shiftl(a, k)   : 2^k * a = remainder * M + c
c = shiftr(a, k)   : a * M / 2^k = c * M + remainder
```

Aus diesen kurzen Funktionen kann man dann die oben angeführten Prozeduren für lange Zahlen zusammensetzen.

Wir fahren fort.
Wir wollen den größten gemeinsamen Teiler zweier Zahlen bestimmen, dafür verwenden wir den Euklidischen Algorithmus.

```
h:= x mod y;
solange h > 0 ist:
x:= y;
y:= x;
h:= x mod y;
ggT(x, y):= y;
```

Eine Variante des Euklidischen Algorithmus liefert für den größten gemeinsamen Teiler g von a und b gleich seine Darstellung als Vielfachsumme: $g = au + bv$.

Das ist natürlich noch nicht der Weisheit letzter Schluß. Wenn die Eingabewerte die Größenordnung N haben, so durchläuft der Euklidische Algorithmus etwa $\log(N)$mal eine Schleife, und jedesmal wird eine Langzahldivision durchgeführt. Der folgende Algorithmus von Lehmer verwendet überwiegend Wort-Divisionen. Die Zahlen `a, b` seien gegeben, die Variablen `ah, bh, A, B, C, D, T, q` sind Worte, `t` und `r` sind lange Zahlen, das Zeichen „/“ bezeichnet die ganzzahlige Division.

```
1. Wenn b < M ist, so verwende den alten Algorithmus.
   Sonst: ah = oberste Stelle von a,
          bh = oberste Stelle von b,
          A:= 1, B:= 0, C:= 0, D:= 1.
2. Wenn bh + C = 0 oder bh + D = 0, so gehe zu 4.
   Sonst: q:= (ah + A)/(bh + C).
   Wenn  q <> (ah + B)/(bh + D) ist, so gehe zu 4.
   (Hier kann ein "Uberlauf auftreten, den man abfangen mu"s.)
3. T:= A - q*C, A:= C, C:= T,
   T:= B - q*D, B:= D, D:= T,
   T:= ah - q*bh, ah:= bh, bh:= T,
   gehe zu 2.
4. Wenn B = 0 ist, so sei t = a mod b, a:= b, b:= t, gehe zu 1.
   (Hier braucht man Langzahlarithmetik, dieser Fall tritt jedoch
   (angeblich) nur mit der Wahrscheinlichkeit 1,4/M = 0,00002 auf.)
   Sonst: t:= A*a, t:= t + B*b, r:= C*a, r:= r + D*b, a:= t, b:= r,
          Gehe zu 1.
```

Eine weitere Variante vermeidet Divisionen (fast) vollständig, es kommen nur Subtraktionen und Verschiebungen vor:

```
1. r:= a mod b, a:= b, b:= r.
   (Hier wird ein einziges Mal dividiert, dieser Schritt kann auch
   weggelassen werden. Von nun an haengt aber die Zahl der Schritte
   nur noch von der Laenge der kleineren der eingegebenen Zahlen ab.)
2. Wenn b = 0 ist, so ist a das Ergebnis.
```

```
   Sonst: k:= 0,
          solange a und b beide gerade sind: k:= k + 1, a:= a/2, b:= b/2.
   (Am Ende haben wir 2^k als genauen Faktor des ggT.)
3. Wenn a gerade ist, so wiederhole a:= a/2, bis a ungerade ist.
   Wenn b gerade ist, so wiederhole b:= b/2, bis b ungerade ist.
4. t:= (a-b)/2
   wenn t = 0 ist, so ist 2^k*a das Ergebnis.
5. Solange t gerade ist, wiederhole t:= t/2.
   Wenn t > 0 ist, so a:= t.
   Sonst: b:= -t.
   Gehe zu 4.
```

Als fünfte Grundrechenart wollen wir die Potenzierung betrachten. Es hat sicher nicht viel Sinn, etwa a^b bilden zu wollen, wobei auch der Exponent b beliebig lang ist, es reicht vielleicht schon $b < 32000$. Wir verwenden einen Trick, der nicht b Multiplikationen erfordert, sondern nur $\log(b)$ Operationen: Wir sehen es am Beispiel

$$a^8 = ((a^2)^2)^2.$$

```
y:= 1;
w:= x;
solange exp > 0 ist:
  wenn exp ungerade ist:
    y:= y * w;
  wenn exp > 1 ist:
    w:= w * w;
    exp:= exp div 2;
dann ist x^exp = y;
```

Auch dieser Algorithmus läßt sich verbessern, zwar nicht in der Anzahl der Rechenschritte, aber in der Größe der Operanden: Im folgenden Algorithmus wird im Schritt 3 immer mit *derselben* Zahl `z` multipliziert.
Um g^n zu berechnen, benötigen wir die Zahl e mit $2^e \geq n < 2^{e+1}$, um diese zu bestimmen, müssen die Bits von n durchmustert werden.

```
1. N:= n, z:= g, y:= z, E:= 2^e.
2. Wenn E = 1 ist, so ist y das Ergebnis.
   Sonst: E:= E/2.
3. y:= y * y
   Wenn N >= E ist, so N:= N - E, y:= y * z.
   Gehe zu 2.
```

Man kann die Division vermeiden, wenn man sich die Bits von `n` anschaut:

```
1. N:= n, z:= g, y:= z, f:= e.
2. Wenn f = 0 ist, so ist y das Ergebnis.
```

```
   Sonst: f:= f - 1.
3. y:= y * y
   Wenn bit(f, N) = 1 ist, so y:= y * z.
   Gehe zu 2.
```

Zum Abschluß wollen wir uns dem Problem der Ein- und Ausgabe langer ganzer Zahlen widmen, einem Fragenkreis, der in den einschlägigen Computeralgebra-Büchern ausgespart bleibt. Wir geben hier PASCAL-Prozeduren an, mit MODULA ist es nicht ganz so einfach. Es versteht sich, daß die „eingebauten“ Prozeduren zum Lesen von 2-Byte- oder Gleitkommazahlen nicht brauchbar sind. Wir lesen also eine Zeichenkette wie '12344444444444444444' und wandeln diese in eine lange Zahl um, dazu lesen wir (von links) Zeichen für Zeichen, wandeln es in eine Zahl um, multiplizieren mit 10 und addieren die nächste Ziffer:

```
procedure readl(var a: IntNumber);
var s: string;
h, zehn: IntNumber;
i: integer;
begin
readln(s);
assic(10, zehn); { zehn := 10 }
a:= NIL;
for i:= 1 to Length(s) do
begin
mult(a, zehn, b);
assic(ord(s[i]) - ord('0'), h);
{ h ist die Zahl, die das Zeichen s[i] darstellt }
wegl(a);
add(b, h, a);
wegl(b);
wegl(h);
end;
end;
```

Das Schreiben ist auch nicht schwierig, wir können aber die Ziffern nicht sofort ausgeben, weil wir sie von rechts nach links berechnen müssen:

```
procedure writel(a: IntNumber);
var s: string; q, r, zehn: IntNumber;  i: integer;
begin
  s:= '';
  while a <> NIL do
  begin
    divmod(a, zehn, q, r);
    a:= q;
    i:= r^.m[1];
```

```
    s:= char(i+ord('0')) + s;
    { das Zeichen der Ziffer i wird an s vorn angeh"angt }
  end;
  write(s);
end;
```

Wir sehen, daß wir bereits zur Ein- und Ausgabe die Rechenoperationen mit langen Zahlen benötigen. Das macht den Test der Arithmetik etwas schwierig, da man sich Zwischenergebnisse, etwa innerhalb der Prozedur `divmod`, nicht mit `writel` ausgeben lassen kann.

13.3 Polynome und „Formelmanipulation“

Wenn man bereits mit ganzen Zahlen rechnen kann, so ist die Implementation der Grundrechenoperationen mit rationalen Zahlen sehr einfach: Eine rationale Zahl ist ein Paar ganzer Zahlen, die wir Zähler bzw. Nenner nennen. Dann gilt

$$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm bc}{bd},\ \frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd},\ \frac{a}{b} : \frac{c}{d} = \frac{ad}{bc}.$$

(Wenn der Nenner verbotenerweise mal gleich Null sein sollte, passiert auch nichts Schlimmes, jedenfalls kein Rechnerabsturz.)
Gelegentlich sollte man einen Bruch kürzen. Dazu bestimmt man den größten gemeinsamen Teiler von Zähler und Nenner und dividert diese durch den ggT.

Wir wollen nun mit Formeln rechnen. Wenn x, y, z Unbestimmte sind, so nennt man einen Ausdruck wie $5x^2 + 7xy + z^5$ ein Polynom. Die Menge aller Polynome in x, y, z mit Koeffizienten aus einem Körper K wird mit $K[x, y, z]$ bezeichnet. Polynome kann man addieren, subtrahieren, multiplizieren.
Wie stellt man Polynome im Rechner dar? Einen Term $x^i y^j z^k$ nennt man ein Monom. Wenn man sich merkt, daß x die 1., y die 2., z die 3. Unbestimmte ist, so ist dieses Monom durch seinen „Exponentenvektor“ $[i, j, k]$ eindeutig bestimmt. Wenn man die Größe der Exponenten beschränkt, so ist die Zahl der möglichen Monome beschränkt, man kann also ein Polynom folgendermaßen abspeichern:
Man numeriert alle möglichen Monome durch (es seien etwa N Stück), dann schafft man sich ein Feld mit N Komponenten und merkt sich zu jedem Monom den entsprechenden Koeffizienten. Wenn ein Monom in einem Polynom gar nicht vorkommt, ist der entsprechende Koeffizient eben gleich Null.
Diese Darstellungsform wird die „dichte“ genannt. Wir wollen uns aber auf die „dünne“ Darstellung beziehen, die nur die wirklich vorkommenden Monome und ihre Koeffizienten speichert.

```
type monom = array[1..max ] of integer;
type polynom = Pointer to polrec;
polrec = Record
c: rat; (* Koeffizient *)
e: monom; (* Exponentenvektor *)
n: polynom;
```

```
end;
```

In der Komponente n (= next) haben wir die Adresse des nächsten Summanden gespeichert, beim letzten Summanden setzen wir $n = \text{NIL}$.
Vernünftig ist es, die Monome nicht zufällig in so eine Liste einzutragen, man ordnet sie am Besten der Größe nach. Wir brauchen also eine Vergleichsrelation für Monome. Der Grad eines Monoms ist die Summe seiner Exponenten, der Grad eines Polynoms ist das Maximum der Grade seiner Monome. Ein Monom soll größer als ein anderes sein, wenn sein Grad größer ist. Für Monome vom Grad 1 legen wir eine Ordnung willkürlich fest, etwa $x > y > z$. Für Polynome gleichen Grades können wir jetzt die lexikographische Ordnung nehmen:

$$x^3 > x^2y > xy^2 > y^3 > x^2z > y^2z > z^3.$$

Diese Grad-lexikographische Ordnung ist nur eine aus einer Vielzahl von Ordnungsmöglichkeiten. Man fordert aber gewöhnlich, daß für Monome p, q, r mit $p > q$ auch stets $pr > qr$ gilt, dies ist hier erfüllt.

Nun können wir Rechenoperationen für Polynome implementieren:

Addition:
Seien zwei Polynome p, q gegeben, in beiden seien die Monome der Größe nach geordnet. Wir schauen uns die jeweils größten Monome an. Es gibt drei Möglichkeiten:

1. $p\hat{}.e > q\hat{}.e$, dann übernehmen wir $p\hat{}.e$ nebst seinem Koeffizienten in $p + q$ und ersetzen p durch $p\hat{}.n$.

2. $p\hat{}.e < q\hat{}.e$, dann übernehmen wir $q\hat{}.e$ nebst seinem Koeffizienten in $p + q$ und ersetzen q durch $q\hat{}.n$.

3. $p\hat{}.e = q\hat{}.e$, dann bilden wir $a = p\hat{}.c + q\hat{}.c$, wenn $a \neq 0$ ist, so übernehmen wir $p\hat{}.e$ mit dem Koeffizienten a in $p + q$ und ersetzen p durch $p\hat{}.n$ und q durch $q\hat{}.n$.

Dies wiederholen wir, bis p und q abgearbeitet sind, also beide auf NIL zeigen.

Die Subtraktion ist analog zu behandeln.

Bei der Multiplikation bekandeln wir zuerst den Spezialfall, wo das Polynom q nur einen Summanden besitzt. Hier ist einfach jeder Summand von p mit $q\hat{}.c$ zu multiplizieren, zu den Exponentenvektoren von p ist $q\hat{}.e$ zu addieren. Da eventuell sehr oft mit derselben Zahl $q\hat{}.c$ zu multiplizieren ist, ist es ratsam, diese Zahl am Anfang zu kürzen.
Den allgemeinen Fall führen wir auf den Spezialfall zurück: Für jeden Summanden m von q bilden wir $m \cdot p$ und addieren all diese Polynome.

Man kann ein Programm schreiben, daß eine eingegebene Zeichenkette in Blöcke zerlegt, die z.B. als Zahlen (12, 1, ...), als Bezeichner von Variablen ($x, x1$, ...), als Funktionswerte ($\sin(x), \sqrt{x}$, ...) usw. interpretiert und Regeln kennt, wie mit diesen Objekten umzugehen ist, etwa $\cos(x + y) = \cos(x)\cos(y) - \sin(x)\sin(y)$. Man kann dann *symbolisch* rechnen, auf numerische Werte kommt es gar nicht an ($x = \sqrt{5}$ wird nicht berechnet, man weiß aber, daß $x^2 = 5$ ist).
Wenn das Programm Differentiations- und Integrationsregeln kennt, so kann man (falls das überhaupt geht), unbestimmte Integrale in geschlossener Form darstellen. Der

Rechner kann in Ruhe alle Tricks durchprobieren. Aus diesem Grund sind Computeralgebrasysteme bei theoretischen Physikern besonders beliebt.

Weit verbreitet sind folgende CA-Systeme, die meist sowohl als UNIX- als auch als DOS-Version verfügbar sind:

- REDUCE von Anthony Hearns, einem Nestor der der Computeralgebra,
- MAPLE,
- DERIVE ist die Fortsetzung von muMath, neben der Software-Version ist es auf einem TI-Taschenrechner „fest verdrahtet“ (wie bekommt man dann ein Update?),
- Mathematica von Stephen Wolfram braucht ein knapp 1000 Seiten dickes Handbuch, und da steht noch gar nicht alles drin (steht in dem Buch), zu den „Packeges“ gibt es nochmals vier Handbücher.
- Macaulay von Stillman, Stillman und Bayer dient vor allem für Rechnungen in der kommutativen Algebra, ebenso
- Singular (Gerd-Martin Greuel, Gerhard Pfister und andere) wurde in Berlin und Kaiserslautern entwickelt; hierfür habe ich die Arithmetik und Matrixoperationen beigesteuert.

Am Montag, dem 5.8.96, erzählte mir Gerhard Pfister, daß in KL der Wert einer 53-reihigen Determinante mit ganzzahligen Einträgen der Größenordnung 10^{20} mittels aller verfügbarer CA-Systeme berechnet wurde. Da keine Software mit Bestimmtheit völlig fehlerfrei ist, kann man sich auf die Korrektheit eines Resultats nur dann verlassen, wenn man es auf unterschiedlichen Systemen verifizieren kann. Die Rechenzeiten der Systeme unterschieden sich heftig:

Maple	500 h
Reduce	80 h
Mathematica	20 h
Singular	0,5 h

Aber auch das Resultat war nicht immer dasselbe: Mathematica hatte ein anderes Ergebnis als die anderen. Natürlich kann kein Mensch das Resultat überprüfen. Die Antwort von Wolfram Research wegen des Fehlers lautete: „Haben Sie nicht ein kleineres Beispiel?“
æ

14 Boolesche Algebren und Boolesche Funktionen

Definition:
Eine Menge B mit drei Operationen $+ : B \times B \longrightarrow B$, $\cdot : B \times B \longrightarrow B$ und $^- : B \longrightarrow B$ sowie zwei ausgezeichnenten Elementen $0, 1 \in B$ heißt Boolesche Algebra, wenn für $a, b, c \in B$ folgende Rechenregeln erfüllt sind:

$a + (b + c) = (a + b) + c, \qquad a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad$ (Assoziativität)

$a + b = b + a, \qquad a \cdot b = b \cdot a \qquad$ (Kommutativität)
$a + a = a, \qquad a \cdot a = a \qquad$ (Idempotenz)
$a + (b \cdot c) = (a + b) \cdot (a + c), \qquad a \cdot (b + c) = a \cdot b + a \cdot c \qquad$ (Distibutivität)
$a + (a \cdot b) = a, \qquad a \cdot (a + b) = a \qquad$ (Absorbtion)
$0 + a = a, \qquad 0 \cdot a = 0$
$1 + a = 1, \qquad 1 \cdot a = a$
$a + \bar{a} = 1, \qquad a \cdot \bar{a} = 0.$

Manchmal schreibt man anstelle von $+$ auch $\vee$ oder $\cup$ und nennt diese Operation Disjunktion, Vereinigung oder Supremum; für $\cdot$ schreibt man dann $\wedge$ oder $\cap$ und nennt es Konjunktion, Durchschnitt oder Infimum. Die Operation $\bar{}$ heißt Komplementierung oder Negation.

Das einfachste Beipiel einer Booleschen Algebra ist die Algebra $\mathbf{B} = \{0, 1\}$, wo sich die Definition der Rechenoperationen schon aus den obigen Regeln ergibt.
Ein weiteres Beispiel ist die Potenzmenge $P(M) = \{U \mid U \subseteq M\}$ einer Menge M mit Durchschnitt und Vereinigung sowie Komplemmentärtmengenbildung als Rechenoperationen. Da, wie man oben sieht, für die Addition und die Multiplikation genau dieselben Rechenregeln gelten, ist es egal, ob wir den Durchschnitt als Addition oder als Multiplikation auffassen.

Wenn B und C Boolesche Algebren sind, so ist $B \times C$ mit komponentenweise definierten Rechenoperationen ebenfalls eine Boolesche Algebra, insbesondere also auch jede kartesische Potenz B^n von B.
Von nun an bezeichne B stets eine Boolesche Algebra.

Für die Komplementierung gelten die folgenden DeMorganschen Regeln:

Satz 14.1 $\overline{x \cdot y} = \bar{x} + \bar{y}$, $\overline{x + y} = \bar{x} \cdot \bar{y}$.

Beweis: Wenn a das Komplement von $x \cdot y$ bezeichnet, so haben wir $a + (x \cdot y) = 1$ und $a \cdot (x \cdot y) = 0$ nachzuweisen:
$(x \cdot y) + (\bar{x} + \bar{y}) = (x + \bar{x} + \bar{y}) \cdot (y + \bar{x} + \bar{y}) = 1 \cdot 1 = 1$,
$(x \cdot y) \cdot (\bar{x} + \bar{x}) = (x \cdot y \cdot \bar{x}) + (x \cdot y \cdot \bar{y}) = 0 + 0 = 0$. Der Beweis der anderen Regel verläuft analog. □

Die soeben benutze Beweismethode („analog“) ist typisch für die Arbeit mit Booleschen Algebren: Man vertauscht die Rechenoperationen miteinander und wendet die analogen Regeln an; dies nennt man „Dualisierung“.

Lemma 14.2 (Kürzungsregel) *Für $x, y, z \in B$ gelte (1) $x \cdot y = x \cdot z$ und (2) $x + y = x + z$. Dann folgt $y = z$.*

Beweis: Zur ersten Gleichung wird sowohl y als auch z addiert:

$$(x \cdot y) + y = (x + y) \cdot (y + y) = (x + y) \cdot y = y$$

nach der Absorbtionsregel; wegen (1) ist dies

$$= (x \cdot z) + y = (x + y) \cdot (z + y),$$

$$(x \cdot z) + z = (x + z) \cdot (z + z) = (x + y) \cdot z = z$$
$$= (x \cdot y) + z = (x + z) \cdot (y + z)$$

und die beiden letzten Terme jeder Gleichung stimmen wegen (2) überein. □

Wir können in B wie folgt eine Ordnung einführen: $a \leq b$ genau dann, wenn $a \cdot b = a$ gilt.

Lemma 14.3 *$a \leq b$ gdw. $a + b = b$.*

Beweis: $b = (a + b) \cdot b = a \cdot b + b \cdot b = a + b$.
Die Umkehrung beweist man surch Vertauschung von a, b und Dualisierung. □

Definition:
Seien $a \leq b \in B$, dann heißt die Menge $\{x \mid a \leq x \leq b\} = [a, b]$ das durch a und b bestimmte Intervall von B.

Wir bemerken, daß $[a, b]$ bezüglich der Addition und Multiplikation abgeschlossen sind. Wenn wir a als Nullelement und b als Einselement auffassen und die Komplementierung in $[a, b]$ relativ zu diesen durchführt (was auch immer das heißen mag), so wird $[a, b]$ wieder eine Boolesche Algebra.

Eine Abbildung zwischen Booleschen Algebren, die mit den jeweiligen drei Rechenoperationen verträglich ist, heißt Homomorphismus Boolescher Algebren. Ein bijektiver Homomorphismus heißt Isomorphismus.

Nun beweisen wir einen Struktursatz, der eine Übersicht über alle endlichen Boolschen Algebren ergibt.

Satz 14.4 *Wenn B eine endliche Boolesche Algebra ist, so gilt $B \cong \mathbf{B}^n$ für eine natürliche Zahl n.*

Beweis: Wir führen die Induktion über $|B|$. Wenn $|B| = 2$ ist, so ist nichts zu zeigen. Sei also die Behauptung für „kleine“ Boolesche Algebren schon bewiesen.
Wir wählen ein Element $a \in B,\ a \neq 0, 1$. Wir setzen

$$X_a = \{(a \cdot b, a + b) \mid b \in B\},$$

dies ist eine Teilmenge von $[0, a] \times [a, 1]$.
Weiter sei $f : B \longrightarrow X_a$ folgende Abbildung: $f(b) = (a \cdot b, a + b)$. Nach der obigen Kürzungsregel ist f injektiv. Wir zeigen die Verträglichkeit mit den Rechenoperationen:

$$f(b \cdot c) = (a \cdot (b \cdot c), a + (b \cdot c)),$$

$$f(b) \cdot f(c) = (a \cdot b, a + b) \cdot (a \cdot c, a + c)$$
$$= (a \cdot a \cdot b \cdot c, (a + b) \cdot (a + c))$$

(komponentenweise Operation)

$$= (a \cdot b \cdot c, a + (b \cdot c)),$$

das ist die Behauptung; die Gleichung

$$f(b+c) = f(b) + f(c)$$

zeigt man analog.
Beim Komplement müssen wir aufpassen: Wir zeigen zunächst, daß $a \cdot \bar{b}$ das Komplement von $a \cdot b$ in $[0, a]$ ist.
$a \cdot \bar{b} + a \cdot b = a \cdot (b + \bar{b}) = a \cdot 1 = a$ ist das größte Element und $(a \cdot \bar{b}) \cdot (a \cdot b) = a \cdot 0 = 0$ ist das kleinste.
Analog: $a + \bar{b}$ ist das Komplement von $a + b$ in $[a, 1]$, da $a + \bar{b} + a + b = 1$ und $(a + \bar{b}) \cdot (a + b) = a + (\bar{b} \cdot b) = a + 0 = a$ ist das kleinste Element.
Nun folgt $f(\bar{b}) = (a \cdot \bar{b}, a + \bar{b}) = \overline{(a \cdot b, a + b)} = \overline{f(b)}$.
Nun ist f auch noch surjektiv, denn für $(x, y) \in [0, a] \times [a, 1]$ setzen wir $b = y \cdot (\bar{a} + x)$, dann ist $f(b) = (a \cdot y \cdot (\bar{a} + x), a + y \cdot (\bar{a} + x)) = (a \cdot y \cdot \bar{a} + a \cdot y \cdot x, (a + y)(a + \bar{a} + x))$; der erste Term ist Null, der zweite wegen $x \leq a \leq y$ gleich x, der dritte Term ist gleich $(a + y) \cdot (a + \bar{a} + x) = (a + y) \cdot 1 = y$.
Also ist f ein Isomorphismus Boolescher Algebren.
Da nun sowohl $[0, a]$ als auch $[a, 1]$ weniger Elemente als B haben, gilt für sie die Induktionsvoraussetzung: $[0, a] \cong \mathbf{B}^k$, $[a, 1] \cong \mathbf{B}^m$, also $B \cong \mathbf{B}^{k+m}$. □

Die Menge $\mathbf{B}^n$ ist isomorph zur Potenzmenge der Menge $\{1, \ldots, n\}$, wir ordnen dem Tupel $(i_1, \ldots, i_n)$ die Menge der k mit $i_k \neq 0$ zu. Dies ist mit den Operationen verträglich.

Folgerung 14.5 (Stonescher Darstellungssatz) *$B \cong P(M)$ für eine endliche Menge M.* □

Folgerung 14.6 *Zwei gleichmächtige endliche Boolsche Algebren (mit 2^n Elementen) sind isomoprh (zu $\mathbf{B}^n$).* □

Wir betrachten nun n-stellige Abbildungen der Form $f : B^n \longrightarrow B$. Wenn f, g zwei solche Abbildungen sind, so können wir $(f \cdot g)(x) = f(x) \cdot g(x)$, $(f + g)(x) = f(x) + g(x)$ und $(\bar{f})(x) = \overline{f(x)}$ setzen und es ist nicht schwer nachzuweisen, daß die Menge $F_n(B) = \{f : B^n \longrightarrow B\}$ so eine Boolsche Algebra wird.

Definition:
Ein Boolsches Polynom in $x_1, \ldots, x_n$ ist folgendes:
(1) $x_1, \ldots, x_n, 0, 1$ sind Boolesche Polynome,
(2) wenn p und q Boolesche Polynome sind, so sind auch $(p) + (q), (p) \cdot (q)$ und $\overline{(p)}$ Boolesche Polynome.

Ein Boolesches Polynom ist also einfach eine Zeichenkette, es gilt $x_1 + x_2 \neq x_2 + x_1$.
Wenn aber $f(x_1, \ldots, x_n)$ ein Boolesches Polynom und B eine Boolesche Algebra ist, so können wir eine Funktion $f^* : B^n \longrightarrow B$ durch $f^*(b_1, \ldots, b_n) = f(b_1, \ldots, b_n)$ konstruieren, indem wir die b_i einfach in f einsetzen und den Wert ausrechnen. Dann gilt natürlich $(x_1 + x_2)^* = (x_2 + x_1)^*$.

Definition:

Zwei Boolsche Polynome f, g heißen äquivalent ($f \sim g$), wenn die zugehörigen Funktionen auf der Algebra $\mathbf{B}$ gleich sind.

Zur Vereinfachung führen wir folgende Schreibweisen ein: $x^1 = x, x^{-1} = \bar{x}$.

Satz 14.7 *Jedes Boolesche Polynom ist äquivalent zu einer „disjunktiven Normalform“*

$$f_d(x_1, \ldots, x_n) = \sum_{i_1,\ldots,i_n \in \{-1,1\}} d_{i_1\ldots i_n} \cdot x_1^{i_1} \cdot \ldots \cdot x_n^{i_n},\ d_{i_1\ldots i_n} \in \{0,1\}.$$

Jedes Boolesche Polynom ist äquivalent zu einer „konjunktiven Normalform“

$$f_k(x_1, \ldots, x_n) = \prod_{k_1,\ldots,k_n \in \{-1,1\}} (k_{i_1\ldots i_n} + x_1^{i_1} + \ldots + x_n^{i_n}),\ k_{i_1\ldots i_n} \in \{0,1\}.$$

Beweis: Es ist $f^*(1^{j_1}, \ldots, 1^{j_n}) = \sum d_{i_1\ldots i_n} 1^{i_1 j_1} \ldots 1^{i_n j_n}$ und ein Summand ist genau dann gleich 1, wenn $i_1 = j_1, \ldots, i_n = j_n$ und $d_{i_1\ldots i_n} = 1$ ist, das heißt, die f_d mit verschiedenen d sind jeweils inäquivalent. Nun ist aber die Anzahl der disjunktiven Normalformen gleich 2^{2^n}, also gleich der Zahl aller Funktionen $\mathbf{B}^n \longrightarrow \mathbf{B}$.
Die zweite Aussage ergibt sich durch Dualisierung. □

Folgerung 14.8 *In der obigen Darstellung ist* $d_{i_1\ldots i_n} = f^*(1^{i_1}, \ldots, 1^{i_n})$.

Beispiel: $f = ((x_1 + x_2) \cdot \bar{x}_1) + (x_2 \cdot (x_1 + \bar{x}_2))$, dann ist $f(0,0) = f(1,0) = 0$ und $f(0,1) = f(1,1) = 1$, die disjunktive Normalform von f erhalten wir, indem wir in der Wertetabelle die Stellen aufsuchen, wo der Wert 1 angenommen wird. Wenn hier ein Argument gleich 0 ist, so ist die entsprechende Variable zu komplementieren, sonst nicht. Also $f \sim \bar{x}_1 x_2 + x_1 x_2$. Dies kann weiter vereinfacht werden: $f \sim (\bar{x}_1 + x_2) \cdot x_2 = 1 \cdot x_2 = x_2$.

Wir überlegen nun, wie man eine Darstellung von Polynomen vereinfachen kann.

Definition Es seien p und q Boolesche Polynome; wir sagen, daß p das Polynom q impliziert, wenn aus $p^*(b_1, \ldots, b_n) = 1$ folgt, daß auch $q^*(b_1, \ldots, b_n) = 1$ gilt (dabei ist $b_i \in \{0,1\}$.
Wir bezeichnen ein Polynom als „Produkt“, wenn es kein +-Zeichen enthält.
Das Polynom p heißt Primimplikant von q, wenn gilt
1) p ist ein Produkt,
2) p impliziert q,
3) kein Teilprodukt von p impliziert q.

Sei zum Beispiel $q = x_1x_2x_3 + x_1\bar{x}_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$ und $p = x_1x_2$, dann wird q von p impliziert, denn $p^* = 1$ gilt nur für $x_1 = x_3 = 1$ und es ist $q^*(1, x_2, 1) = (x_2 + \bar{x}_2 + \bar{x}_2)^* = 1$, aber z.B. x_1 impliziert q nicht, da $q^*(1, x_2, x_3) = (x_2x_3 + \bar{x}_2x_3 + 0)^* = (x_2 + \bar{x}_2)x_3 = x_3 \neq 1$ ist.

Wir bemerken, daß ein Produkt genau dann 1 ist, wenn alle nichtkomplementierten Variablen gleich 1 und alle komplementierten Variablen gleich 0 gesetzt werden.
Alle Summanden einer disjunktiven Normalform sind Implikanten.

Satz 14.9 *Jedes Polynom ist äquivalent zur Summe seiner Primimplikanten.*

Beweis: Seien $p_1, \ldots, p_m$ die Primimplikanten von q, wir setzen $p = p_1 + \ldots + p_m$. Sei nun $p^*(b_1, \ldots, b_n) = 1$, dann gibt es ein p_i mit $p_i(b_1, \ldots, b_n) = 1$ und da p_i das Polynom q impliziert, gilt auch $q^*(b_1, \ldots, b_n) = 1$.
Sei umgekehrt $q^*(b_1, \ldots, b_n) = 1$, wir setzen $s = x_1^{i_1} \cdots x_n^{i_n}$ mit $i_k = 1$, falls $b_k = 1$ und $i_k = -1$ für $b_k = 0$, dann ist s ein Implikant von q. Wir lassen nun aus dem Term s alle die x_i weg, für die $q^*(b_1, \ldots, b_{i-1}, \bar{b}_i, \ldots) = 1$ ist; das Ergebnis sei r. Dann gilt: r impliziert q, aber kein Teilwort von r impliziert q, folglich ist r als Primimplikant gleich eienm der p_j, also folgt $p^*(b_1, \ldots, b_n) = 1$, d.h. $p \sim q$. □

Von der disjunktiver Normalform eines Polynoms ausgehend kann man eine Darstellung als Summe von Primimplikanten erhalten, indem man für alle Paare von Summanden, wo dies möglich ist, die Regel $px + p\bar{x} \sim p$ anwendet.