GMD Report
97

GMD –
Forschungszentrum
Informationstechnik
GmbH

Gabriele Reich (Hrsg.)

Bundeswettbewerb Informatik

Aufgaben und Lösungen

12. Wettbewerb 1993/1994

13. Wettbewerb 1994/1995

Mai 2000

© GMD 2000

GMD –
Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
http://www.gmd.de

In der Reihe GMD Report werden Forschungs- und Entwicklungsergebnisse zum wissenschaftlichen, nichtkommerziellen Gebrauch veröffentlicht. Jegliche Inhaltsänderung des Dokuments sowie die entgeltliche Weitergabe sind verboten.

The purpose of the GMD Report is the dissemination of research work for scientific non-commercial use. The commercial distribution of this document is prohibited, as is any modification of its content.

Anschrift der Herausgeberin/Address of the editor:
Gabriele Reich
c/o Bundeswettbewerb Informatik
Ahrstraße 45
D-53175 Bonn
E-mail: bwinf@bwinf.de
www.bwinf.de

Mitglieder und aktive Gäste des Aufgabenausschusses während des 12. und 13. Bundeswettbewerbs Informatik:
Johannes Buchmann, Michael Fothe, Friedrich Gasper, Ursula Goltz, Rüdiger Hüttenhain, Herbert Klaeren, Ina Leiß, Volker Neumann, Lothar Oppor, Gabriele Reich, Vera Reineke (Vorsitzende), Kirsten Riechmann, Kirsten Schlüter.

An diesem Band haben außerdem mitgearbeitet:
Johannes Busse, Thomas Fischbacher, Thomas Jansen

ISSN 1435-2702

Zusammenfassung

Obwohl für viele Kinder und Jugendliche der Einstieg in die Welt der Informatik mit mehr oder weniger geistreichen Computerspielen beginnt, so ist die Informatik doch deutlich mehr. Den Schülerinnen und Schülern dies zu vermitteln ist ein Anliegen des Bundeswettbewerbs Informatik.

Von den Teilnehmerinnen und Teilnehmern werden zu Beginn jedes Wettbewerbs stets die gleichen Fragen gestellt, was denn ein Programmablaufprotokoll sei, wieviel Kommentar im Programm selbst erforderlich sei usw. Und natürlich interessiert auch, bei allem Selbstvertrauen in die eigene Lösung, wie gute Lösungen zu den im Wettbewerb gestellten Aufgaben aussehen könnten. Diese Fragen versucht das vorliegende Buch zu beantworten. Gleichzeitig soll es Informatik-Lehrkräften Anregungen und Beispiele für den Unterricht geben. Außerdem kann es auch für solche Jugendliche als Einstieg in die Informatik dienen, die zwar schon eine Programmiersprache beherrschen, aber noch nicht wissen, wie sie diese sinnvoll einsetzen können.

Aushängeschild des Wettbewerbs sind die Aufgaben, die jährlich in der ersten und zweiten Runde zu lösen sind. Die Aufgaben stammen aus verschiedenen Problemfeldern der Informatik. Sie sollen möglichst viele Jugendliche ansprechen und zusätzlich zur Teamarbeit anregen.

Dieses Buch enthält die Erstrunden- und Zweitrundenaufgaben des 12. und des 13. Bundeswettbewerbs Informatik mit Lösungen. Letztere werden sowohl als natürlichsprachlich formulierte Idee als auch in Form eines Programms angegeben.

Schlagwörter: Informatik, Wettbewerb, Bundeswettbewerb Informatik, Aufgaben, Programmieren, Schulinformatik, Informatikunterricht

Abstract

When children and young people get in touch with computers for the first time, they will probably play a more or less reasonable computer game. But dealing with computers, and especially with computer science, is more than that. By participating in "Bundeswettbewerb Informatik" (the German annual computer science contest for high school students) young people can experience that.

Contestants, of course, are very eager to find a good solution to the problems posed. But what should a good solution look like? What about the documentation of programs and program runs? And, what about a really good program that solves a given problem? This book would like to give answers to these questions. At the same time, it is also aimed at computer science teachers who may take some inspiration and examples from it. Last but not least, it may offer first insights into computer science to students who are able to write programs but do not yet know to make the best of this ability.

The image and reputation of "Bundeswettbewerb Informatik" is dominated by the problems that are to be solved in the first and second rounds of the contest. These problems originate from different areas of computer science. It is their purpose to attract as many students as possible and (in the first round) to make contestants form teams to cooperate.

This book contains the problems of the first and secound rounds of the 12th and 13th Bundeswettbewerb Informatik. To each problem, a solution is presented: a natural language description of the general idea and a program which implements this idea.

Keywords: computer science, contest, Bundeswettbewerb Informatik, problems, programming, computer science at school, computer science lessons

Inhaltsverzeichnis

1 Einleitung **7**

1.1 Ablauf . . . 7

1.2 Organisation . . . 9

1.3 Statistik . . . 10

1.4 Inhalt des Buches . . . 10

1.5 Interpretation der Aufgabentexte . . . 11

1.6 Dokumentationsrichtlinien . . . 11

1.7 Musterlösungen . . . 12

2 12. Wettbewerb: Die Aufgaben der 1. Runde **13**

2.1 Skyline . . . 13

2.2 Hast Du Töne? . . . 21

2.3 Hotelschlüssel . . . 28

2.4 Das Haus des Nikolaus . . . 39

2.5 Das Geheimnis der 3 Karten . . . 52

3 12. Wettbewerb: Die Aufgaben der 2. Runde **57**

3.1 Mustererkennung . . . 57

3.2 Hamiltons Roboter . . . 65

3.3 Pentapuzzle . . . 71

3.4 Kuriositäten und besonders nette Formulierungen . . . 77

4 13. Wettbewerb: Die Aufgaben der 1. Runde **78**

4.1 Schiffe versenken . . . 78

4.2 Nochmal Schiffe versenken . . . 85

4.3 Rätselecke . . . 86

4.4 Verschlüsselung . . . 91

4.5 Rotieren nach Plan 93
4.6 Würfelspiel 95
4.7 Kuriositäten und besonders nette Formulierungen 102

5 13. Wettbewerb: Die Aufgaben der 2. Runde 103

5.1 Promesiphus 103
5.2 Testreiheneditor 114
5.3 Dreiecksflug 121
5.4 Kuriositäten und besonders nette Formulierungen 126

1 Einleitung

Obwohl für viele Kinder und Jugendliche der Einstieg in die Welt der Informatik mit mehr oder weniger geistreichen Computerspielen beginnt, so ist die Informatik doch deutlich mehr. Den Schülerinnen und Schülern dies zu vermitteln, ist ein Anliegen des Bundeswettbewerbs Informatik.

Von den Teilnehmerinnen und Teilnehmern werden zu Beginn jedes Wettbewerbs stets die gleichen Fragen gestellt, was denn ein Programmablaufprotokoll sei, wieviel Kommentar im Programm selbst erforderlich sei usw. Und natürlich interessiert auch, bei allem Selbstvertrauen in die eigene Lösung, wie gute Lösungen zu den im Wettbewerb gestellten Aufgaben aussehen könnten. Diese Fragen versucht das vorliegende Buch zu beantworten. Gleichzeitig soll es Informatik-Lehrkräften Anregungen und Beispiele für den Unterricht geben. Außerdem kann es auch für solche Jugendliche als Einstieg in die Informatik dienen, die zwar schon eine Programmiersprache beherrschen, aber noch nicht wissen, wie sie diese sinnvoll einsetzen können.

"Aushängeschild" des Wettbewerbs sind die Aufgaben, die jährlich in der ersten und zweiten Runde zu lösen sind. Die Aufgaben stammen aus verschiedenen Problemfeldern der Informatik. Sie sollen möglichst viele Jugendliche ansprechen und zusätzlich zur Teamarbeit anregen.

1.1 Ablauf

Der Bundeswettbewerb Informatik gehört zu den gesamtstaatlich geförderten Schülerwettbewerben der Bundesrepublik Deutschland und steht unter der Schirmherrschaft des Bundespräsidenten. Er wird jährlich in drei Runden durchgeführt.

1. Runde: Es werden fünf oder sechs allgemeinverständliche Aufgaben gestellt. Diese werden im September an alle Gesamtschulen, Gymnasien und Berufsschulen in den Ländern der Bundesrepublik geschickt. Außerdem können die Aufgabenblätter bei der Geschäftsstelle des Wettbewerbs angefordert werden. Einsendeschluß für die 1. Runde ist in der ersten Novemberhälfte. Um an der zweiten Runde teilnehmen zu können, müssen die Jugendlichen mindestens drei Aufgaben weitgehend richtig lösen, wobei auch Gruppenarbeit erlaubt ist. Teilnahmeberechtigt sind Jugendliche bis zu 21 Jahren, sofern sie noch nicht im Beruf stehen oder studieren. Die Einsendungen werden von Studierenden der Informatik nach genauen Vorgaben des Aufgabenausschusses korrigiert.

2. Runde: Wer die erste Runde allein oder im Team erfolgreich bestanden hat, erhält mit der Auswertung der ersten Runde die drei Aufgaben der zweiten Runde. Diese Aufgaben sind schwieriger, erfordern mehr Informatikkenntnisse und sind "nach oben offen", d. h. jeder Teilnehmer sollte jede Aufgabe nach eigener Einschätzung erweitern und auf diese Weise Zusatzpunkte erwerben. Wegen der notwendigen Einzelbewertung sind Teams in der zweiten Runde nicht zugelassen. Einsendeschluß ist üblicherweise Ende April. Die Bewertung der Einsendungen erfolgt im Mai nach dem schon in der ersten Runde angewandten Verfahren.

Endrunde: Die etwa 30 Jugendlichen, die die beste Bewertung in der zweiten Runde erreicht haben, werden zum Abschlußkolloquium der dritten Runde eingeladen. Dieses dauert zwei Tage, zuzüglich Anreise am Vortag und der Siegerehrung am dritten Tag. In diesen zwei Tagen bearbeiten die Jugendlichen in Gruppen von fünf bis sechs Personen verschiedene, recht schwierige Aufgaben, zu denen in der Regel keine effizienten Lösungsverfahren bekannt sind. Daher kommt es darauf an, wie die Jugendlichen neue Fragestellungen in Angriff nehmen und welche Informatikmethoden sie dafür einsetzen. Außerdem finden Einzelgespräche von 30 Minuten Dauer mit Fachleuten aus Hochschule und Schule statt. An den Nachmittagen werden die Ergebnisse der Gruppen im Plenum vorgestellt und diskutiert. Die Fachleute beobachten die Diskussionen und Lösungsansätze der Jugendlichen und erhalten hierbei ein recht klares Bild über die Befähigungen jedes einzelnen sowohl bezüglich der Einzelleistung als auch bezüglich der Gruppenarbeit. Die Auswahlkommission, die aus den genannten Fachleuten besteht, wählt in einer Klausursitzung am Abend des zweiten Tages die Bundessieger aus. Der dritte Tag gehört der Siegerehrung, an der Vertreter des Bundesministers für Bildung und Forschung und des Präsidenten der Kultusministerkonferenz teilnehmen.

Die Endrunde findet im Herbst statt und wird von einem privaten Wirtschaftsunternehmen ausgerichtet, das die Organisation und die anfallenden Kosten übernimmt. Der Bundeswettbewerb ist bemüht, auf lange Sicht alle Regionen Deutschlands bei der Wahl der Durchführungsorte möglichst gleichmäßig zu berücksichtigen.

Preise: Die Teilnahme an jeder der drei Runden wird durch eine Urkunde bestätigt. Für die Teilnehmer der dritten Runde gibt es Sach- und zum Teil Geldpreise. Die Bundessieger werden in der Regel ohne weiteres Aufnahmeverfahren in die Studienstiftung des deutschen Volkes aufgenommen.

1.2 Organisation

Träger des Wettbewerbs:

- Gesellschaft für Informatik e.V. (GI), Wissenschaftszentrum, Ahrstraße 45, 53175 Bonn,
  etwa 18.000 Mitglieder.
  Präsident 1993: Prof. Dr. Roland Vollmar, Universität Karlsruhe
  1994/95: Prof. Dr. Wolfgang Glatthaar, IBM
- GMD – Forschungszentrum Informationstechnik GmbH, Schloß Birlinghoven, 53754 Sankt Augustin,
  Großforschungseinrichtung des Bundes mit Beteiligung der Länder Hessen und Nordrhein-Westfalen.
  Zuständiges Vorstandsmitglied bis Ende 1995:
  Dipl.-Kaufm. Friedrich Winkelhage

Gremien des Wettbewerbs: (besetzt durch die Träger des Wettbewerbs)

- Lenkungsgremium: Auswahlausschuß
  etwa acht Mitglieder,
  Vorsitz 1993/94: Prof. Dr. Rüdiger Loos, Universität Tübingen
  1995: Prof. Dr. Rainer Busch, Fachhochschule Bochum.
- Aufgabenausschuß:
  8 – 10 Mitglieder, Vorsitz: OStR Vera Reineke, Hannover.
- Für die Endrunde werden jedes Jahr 12 – 15 unabhängige Fachleute benannt, die die Auswahlkommission zur Ermittlung der Bundessieger bilden.

Geschäftsführung:

Dr. Gabriele Reich
Wissenschaftszentrum
Ahrstraße 45
53175 Bonn
Tel.: 0228 – 302 197
email: bwinf@bwinf.de
WWW: http://www.bwinf.de

Sekretariat: Rainer Köhlen

Fax: 0228 – 37 29 000

Die Kosten für den Wettbewerb (ohne die Endrunde) werden vom Bundesministerium für Bildung und Forschung, von der GI und von der GMD getragen.

1.3 Statistik

Die ersten drei Wettbewerbe waren ähnlich wie "Jugend forscht" organisiert und wurden von Prof. Dr. Volker Claus, dem ersten Vorsitzenden des Auswahlausschusses, im Auftrag der Gesellschaft für Informatik durchgeführt. Seit 1985/86 läuft der Wettbewerb nach dem beschriebenen Schema ab.

Nr.	Jahr	Anzahl Teilnehmer in 1. Rd.	2. Rd.	3. Rd.	Bundes-sieger	Ausrichtende Firma mit Datum und Ort der Siegerehrung
1.	80/81	113	–	–	1	IFIP, Lausanne, 31.7.81
2.	82/83	221	–	–	3	GMD, St. Augustin, 1.6.83
3.	84/85	150	–	–	4	Nixdorf, Paderborn, 10.5.85
4.	85/86	1109	246	28	7	Siemens, München, 13.9.86
5.	86/87	2306	328	32	6	Daimler Benz, Stuttgart, 16.10.87
6.	87/88	1449	236	30	6	Colonia Versicherung, Köln, 22.9.88
7.	88/89	1770	386	30	6	Mannesmann Kienzle, Villingen, 6.10.89
8.	89/90	1783	271	39	8	IBM, Hannover, 19.10.90
9.	90/91	3069	289	30	6	DEC, München, 26.9.91
10.	91/92	1937	306	34	8	Jenoptic und Universität, Jena, 17.9.92
11.	92/93	2008	410	36	5	Telekom und Universität, Leipzig, 16.9.93
12.	93/94	1583	230	32	7	Drägerwerk AG, Lübeck, 25.8.94
13.	94/95	1384	241	32	6	Sun Microsystems, Telekom, Bonn, 4.10.95

1.4 Inhalt des Buches

Zu den Aufgaben der 1. Runden geben wir Lösungsideen, das Grundgerüst des Algorithmus auf hochsprachlichem Niveau, den Programmtext und je nach Aufgabe einige weitere Erläuterungen an. Zu den Aufgaben der 2. Runden drucken wir keine Programmtexte ab, denn diese sind in der Regel so umfangreich, daß sich wohl kein Leser die Mühe machen würde, den Text Anweisung für Anweisung nachzuvollziehen. Stattdessen erläutern wir, welche Überlegungen zu den Aufgabenstellungen und Lösungen sinnvoll sind. Die Aufgabenstellungen drucken wir unverändert ab, ohne Änderung solcher Formulierungen, die sich inzwischen als mehrdeutig herausgestellt haben.

Die meisten Programmtexte dieses Buches sind in Pascal angegeben. Dies liegt daran, daß Pascal nicht nur weit verbreitet, sondern auch didaktisch gut geeignet ist. Außerdem ist die Mehrzahl der Einsendungen zum Bundeswettbewerb Informatik in Pascal programmiert. Die Analyse der verwendeten Programmiersprachen ist übrigens recht interessant: Viele Jahre war die meistverwendete Sprache Basic. Inzwischen ist es Pascal, Basic stirbt langsam aus, dafür mischen sich einige Lösungen in C und C++, aber auch in Java und in Delphi unter die Einsendungen. Wo es uns sinnvoll erschien, haben auch wir Lösungen in anderen Programmiersprachen aufgenommen. Wir haben versucht, möglichst Original-

Programme als Beispiellösungen zu verwenden, wie sie von den Teilnehmern der Wettbewerbe eingesandt wurden. Nur dort, wo die Programme zu weit von unseren stilistischen Empfehlungen abwichen, haben wir sie vorsichtig modifiziert. Für sachliche Fehler, die sich bei diesem Vorgang eventuell eingeschlichen haben, bitten wir um Entschuldigung.

1.5 Interpretation der Aufgabentexte

Bei aller Sorgfalt, mit der die Mitglieder des Aufgabenausschusses die Aufgabentexte formulieren, lassen sich Unklarheiten, Mißverständnisse und Tippfehler leider nie völlig ausschließen. Wir wollen daher an dieser Stelle kurz erläutern, wie in solchen Fällen zu verfahren ist.

Ganz wichtig ist, zunächst den unklaren Aufgabentext in aller Ruhe und Sorgfalt nochmals durchzulesen und dann den gesunden Menschenverstand zu benutzen. Mitunter hilft es auch schon, sich den Sinn der verwendeten Wörter klarzumachen. Wenn es heißt, das Programm P *braucht* nur a oder b zu tun, dann genügt es völlig, wenn P auch nur a oder b tut, andererseits ist es aber natürlich kein Fehler, wenn P noch mehr kann. Sonst hätte es nämlich heißen müssen, P *darf* nur a oder b tun.

Insbesondere bei den Aufgaben der 2. Runde ist es durchaus erlaubt und erwünscht, unter mehreren Interpretationsmöglichkeiten diejenige zu wählen, die einem selbst auch bei mehrmaligem Nachdenken am sinnvollsten erscheint. Dabei kann man beispielsweise folgende Überlegungen anstellen: Hat die Aufgabe einen Bezug zur Realität? Wenn ja, wie ist die reale Situation? Kann ich aus der Realität die fehlenden Informationen gewinnen oder ist die Realität so komplex, daß ich vereinfachen muß? Die schließlich gefällten Entscheidungen müssen selbstverständlich dokumentiert werden, eventuell auch mit einer kurzen Begründung, warum man sich gerade so entschieden hat, damit die Korrektoren den Lösungsansatz nachvollziehen können.

1.6 Dokumentationsrichtlinien

Grundsätzlich gilt: Die Dokumentation soll so beschaffen sein, daß jemand, der nur die Aufgabenstellung kennt, anhand der Dokumentation (einschließlich Kommentaren im Programmtext) die Lösung nachvollziehen und den Aufbau des Programms und das Zusammenspiel der einzelnen Programmteile verstehen kann. Das heißt aber nicht, daß Trivialitäten in epischer Breite geschildert werden sollen, sondern eine knappe, aber präzise Darstellung ist erwünscht.

Eine gute Dokumentation wird natürlich nicht nur verlangt, damit die Korrektoren die Einsendungen bewerten können, sondern in erster Linie deshalb, weil es im täglichen Leben sehr häufig vorkommt, daß man das Programm eines anderen verstehen muß. Und ein weitgehend unkommentiertes und undokumentiertes Programm zu verstehen, ist bereits dann schwierig, wenn man es selbst geschrieben hat.

1.7 Musterlösungen

Gegen Einsendung einer formatierten 3,5"DOS-Diskette und eines frankierten und adressierten Rückumschlages senden wir die Programme zu, die in den bisher erschienenen Büchern zum Wettbewerb abgedruckt wurden.

2 12. Wettbewerb: Die Aufgaben der 1. Runde

2.1 Skyline

2.1.1 Aufgabenstellung

Der Rat des neuen Stadtstaates Utopolis möchte ein Stadtwappen einführen, welches unter anderem stilisiert, aber maßstabsgerecht die Silhouette der Stadt enthält. Alle hohen Gebäude der Stadt sind auf der Ebene errichtete Quader. Jemand hat sich die Mühe gemacht, die Koordinaten der linken und der rechten oberen Ecke aller Gebäude der Stadt in einem 80×25-Raster zu bestimmen und aufzuschreiben. Beispiel siehe unten.

Aufgabe:
Hilf dem Stadtrat, indem Du ein Programm schreibst, welches die ungeordnete Punktfolge der linken und rechten Ecken einliest, die Skyline als geordnete Punktfolge ausgibt und die Rechtecke und die Skyline als Graphik druckt (Textzeichen für die Graphik sind erlaubt). Die Eingabe-Punktfolge soll auf Zulässigkeit geprüft werden und eventuell zurückgewiesen werden.

Im Beispiel wird die Skyline der ersten beiden Hochhäuser ausgegeben als Punktfolge (13,0), (13,14), (19,14), (19,9), (23,9), (23,0).

Schicke uns fünf Ergebnisse des Programms, darunter das zu der Punktfolge:
(6,9), (16,9), (72,16), (79,16), (7,12), (13,12), (9,14), (11,14), (17,7), (24,7), (19,4), (22,4), (17,10), (22,10), (24,7), (28,7), (31,12), (39,12), (42,15), (57,15), (41,15), (44,15), (54,12), (60,12), (58,5), (60,5), (35,8), (43,8)

Beispiel:
(17,9), (23,9), (13,14), (19,14), (46,10), (52,10), (37,23), (55,23), (34,3), (38,3)

2.1.2 Lösungsidee

Es sind drei Teilaufgaben zu lösen:

1. Einlesen der Koordinaten,
2. Ausgabe der Koordinaten der Skyline,
3. Drucken der Hochhäuser und der Skyline.

Als Datenstruktur verwenden wir für die Skyline ein Array *Skyline* der Größe 200, in dem die Eckpunkte (jeweils x- und y-Wert) eingetragen werden (es handelt sich also um ein zweidimensionales Array[1..200,1..2]; da es für jeden x-Wert höchstens zwei Skyline-Eckpunkte geben kann, ist die Arraygröße 200 auf jeden Fall ausreichend). Für die Hochhäuser verwenden wir keine eigene Datenstruktur, sondern wir definieren ein zweidimensionales Array *PixelStatus [0..80,0..25]*, in dem vermerkt wird, ob an der entsprechenden Stelle nichts ist (PixelStatus = *LEER*), ob sich dort der Rand eines Hochhauses bzw. der Horizont befindet (PixelStatus = *RAND*) oder ob sich dort ein Haus befindet (PixelStatus = 1..14, d. h. es können sich höchstens 14 Häuser überlappen; dann kommt man mit insgesamt 16 Farben aus).

Bei der Eingabe werden direkt drei Werte, jeweils getrennt durch ein Leerzeichen eingegeben. Diese Werte sind die x-Koordinaten der linken oberen und der rechten oberen Hochhausecke und die y-Koordinate der linken oberen Hochhausecke. Die y-Koordinate der rechten oberen Ecke wird nicht mehr eingegeben, sondern von der linken Ecke übernommen. Bei der Eingabe der Koordinaten wird jeweils geprüft, ob sie im zulässigen Bereich liegen. Wenn nicht, wird die Eingabe zurückgewiesen. Die Eingabe ist beendet, wenn alle eingegebenen Werte 0 sind.

Nach der Eingabe dreier korrekter Koordinaten wird das zugehörige Haus in das Array *PixelStatus* eingetragen. Dabei werden die untere, obere, rechte und linke Begrenzung des Hauses auf *RAND* gesetzt, der Wert der dazwischenliegenden Pixel wird jeweils um 1 erhöht. War das Feld vorher leer, bekommt es also den Wert 1, stehen an der betreffenden Stelle bereits drei Häuser, so bekommt es den Wert 4. Der Wert eines *RAND*-Pixels bleibt unverändert.

Die Pixel der Rechtecke werden gezeichnet, indem für jedes Pixel die Prozedur *Punkt* aufgerufen wird, die an der entsprechenden Stelle auf dem Bildschirm ein kleines Rechteck zeichnet. Dabei werden für verschiedene Werte von *PixelStatus* auch verschiedene Farben verwendet, so daß sichtbar wird, wo mehrere Häuser voreinander stehen. Die Ränder werden weiß gezeichnet.

Die Skyline wird berechnet, indem die Pixel abgelaufen werden, die die äußeren Ränder von Häusern sind. Dies kann man durch Abfragen realisieren, auf wel-

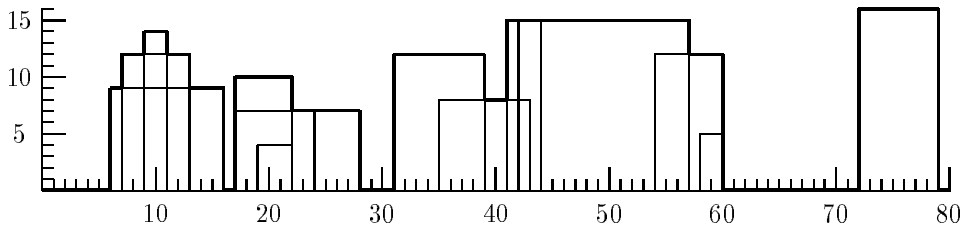
cher x- bzw. y-Koordinate man sich gerade befindet, aus welcher Richtung man gekommen ist und welche benachbarten Pixel den Wert *RAND* haben. Jeder Punkt, an dem sich die Richtung der Skyline ändert, wird in das Array *Skyline* eingetragen.

Die Ausgabe der Skyline als geordnete Punktfolge besteht dann nur darin, das Array *Skyline* der Reihe nach aufzulisten.

Beim Zeichnen der Skyline werden die in *Skyline* gespeicherten Punkte miteinander verbunden und wiederum durch die Prozedur *Punkt* auf dem Bildschirm gezeichnet.

Beim Zeichnen der Rechtecke und der Skyline wird in den Hintergrund ein Punktraster mit Achsenbeschriftungen gelegt (Prozedur *Koordinatensystem*), so daß die x- und y-Koordinaten der Häuser bzw. der Skyline am Bildschirm abgelesen werden können.

Die Lösung zum geforderten Beispiel:



2.1.3 Programmtext

```
program SkylineVonUtopolis;

uses crt, graph;

const bgipfad = 'c:\bp\bgi';
        LEER = 0;
        RAND = 15;   { Werte für PixelStatus:
                       LEER wenn unbesetztes Feld
                       RAND wenn Hochhaus-/Skylinerand
                       1..14 Anzahl der sich schneidenden Häuser }
```

```
var PixelStatus : array [0..80] of array [0..25] of integer;
                { Status der einzelnen Felder, s. o. }
        Skyline : array [1..200] of array [1..2] of integer;
                { Die Punkte der Skyline (jeweils x-/y-Koordinate }
        SkylinePunkte : integer;
                { Anzahl der Punkte der Skyline }


procedure Initialisieren;
  var x, y : integer;
  begin
    for x := 0 to 80 do
      for y := 0 to 25 do
        PixelStatus [x,y] := 0;
  end;

procedure EingabeLesen;

  procedure HausEintragen (x1, x2, y : integer);
    var i, j : integer;
    begin
      for i := x1 to x2 do
        begin
          PixelStatus [i, 0] := RAND;
          PixelStatus [i, y] := RAND;
        end;
      for j := 1 to y-1 do
        begin
          PixelStatus [x1, j] := RAND;
          PixelStatus [x2, j] := RAND;
        end;
      for i := x1+1 to x2-1 do
        for j := 1 to y-1 do
          if (PixelStatus [i, j] < 14) then
            inc (PixelStatus [i, j]);
    end;  {HausEintragen}

  var x1, x2, y : integer;
  begin
    clrscr;
    writeln ('SKYLINE');
    writeln;
```

```
    writeln ('Bitte␣geben␣Sie␣die␣Koordinaten␣der␣oberen');
    writeln ('Begrenzung␣der␣Hochhäuser␣im');
    writeln ('Format␣x1␣x2␣y␣ein.');
    writeln ('Beispiel:␣70␣75␣20;␣0␣0␣0␣beendet␣die␣Eingabe');
    writeln;
    repeat
      write ('>␣');
      readln (x1, x2, y);
      if (x1 = 0) and (x2 = 0) and (y = 0) then
        break;
      if (x1 < 0) or (x2 > 80) or (x2 <= x1) or (y < 1)
          or (y > 25) then
        writeln ('Ungültige␣Eingabe!')
        else HausEintragen (x1, x2, y);
    until false;
  end;  {EingabeLesen}

procedure Koordinatensystem;
  var i, j : integer;
        s : string;
  begin
    for i := 0 to 80 do
      for j := 0 to 25 do
        putpixel (77 + ( i * 5 ), 298 - (j * 5), 15);
    for i := 0 to 16 do
      begin
        str ((i * 5):2, s);
        outtextxy (70 + (i * 25), 316, s);
        line (77+25*i, 303, 77+25*i, 312);
      end;
    for i := 0 to 5 do
      begin
        str ((i * 5):2, s);
        outtextxy (44, 294 - (i * 25), s);
        line (72, 298-25*i, 63, 298-25*i);
      end;
  end;  {Koordinatensystem}

procedure Punkt (x, y, farbe : integer);
  begin
    setfillstyle (1, farbe);
    bar (75+5*x, 300-y*5, 79+5*x, 296-y*5);
  end;  {Punkt}
```

```
procedure RechteckeZeichnen;
  var i, j, gd, gm : integer;
  begin
    gd := Detect;
    InitGraph (gd, gm, bgipfad);
    outtextxy (10, 10, 'Die␣Rechtecke:');
    for i := 0 to 80 do
      for j := 0 to 25 do
        if (PixelStatus [i, j] <> 0) then
          Punkt (i, j, PixelStatus [i, j]);
    Koordinatensystem;
    while (readkey = #0) do;
    closegraph;
  end;  {RechteckeZeichnen}

procedure SkylineBerechnen;      { durch Abschreiten des äußeren Randes }
  const HOCH = 1;
        RECHTS = 2;
        RUNTER = 3;                          { Konstanten für Richtungen }
  var i, j, x, y, ar, nr : integer;          { ar, nr : alte/neue Richtung }
  begin
    for i := 0 to 80 do
      PixelStatus [i, 0] := RAND;        { Der Horizont gehört zur Skyline }
    SkylinePunkte := 0;
    x := 0;
    y := 0;
    nr := 0;
    repeat
      ar := nr;
      repeat
        Skyline [SkylinePunkte+1, 1] := x;
        Skyline [SkylinePunkte+1, 2] := y;
        nr := 0;
        if (y > 0) and (PixelStatus [x, y-1] = RAND) then
          nr := RUNTER;
        if (x < 80) and (PixelStatus [x+1, y] = RAND) then
          nr := RECHTS;
        if (y < 25) and (ar <> RUNTER)
            and (PixelStatus [x, y+1] = RAND) then
          nr := HOCH;
```

```
        case nr of
          HOCH : inc (y);
          RECHTS : inc (x);
          RUNTER : dec (y);
        end;
      until (nr <> ar) or ((x = 80) and (y = 0));
      if (nr <> ar) then
        inc (SkylinePunkte);
    until (x = 80) and (y = 0);
    inc (SkylinePunkte);
    Skyline [SkylinePunkte, 1] := 80;
    Skyline [SkylinePunkte, 2] := 0;
  end;  {SkylineBerechnen}

procedure SkylineAusgeben;
  var i, j : integer;
  begin
    clrscr;
    writeln ('Die␣Skyline␣als␣geordnete␣Punktfolge:');
    writeln;
    for i := 1 to SkylinePunkte do
      begin
        if (i <> 1) then write (',␣');
        write ('(', Skyline [i, 1]:2, ',␣' ,Skyline [i, 2]:2, ')');
      end;
    while (readkey = #0) do;
  end;  {SkylineAusgeben}

procedure SkylineZeichnen;
  var i, j, gd, gm : integer;
  begin
    gd := Detect;
    InitGraph (gd, gm, bgipfad);
    outtextxy (10, 10, 'Die␣Skyline:');
    for i := 1 to SkylinePunkte-1 do
      begin
        if (Skyline [i, 2] = Skyline [i + 1, 2]) then
          for j := Skyline [i, 1] to Skyline [i + 1, 1] do
            Punkt (j, Skyline [i, 2], 4);
        if (Skyline [i, 2] < Skyline [i + 1, 2]) then
          for j := Skyline [i, 2] to Skyline [i + 1, 2] do
            Punkt (Skyline [i, 1], j, 4);
```

```
        if (Skyline [i, 2] > Skyline [i + 1, 2]) then
            for j := Skyline [i + 1, 2] to Skyline [i, 2] do
                Punkt (Skyline [i, 1], j, 4);
      end;
    Koordinatensystem;
    while (readkey = #0) do;
    closegraph;
  end;  {SkylineZeichnen}

begin   {SkylineVonUtopolis}
  Initialisieren;
  EingabeLesen;
  RechteckeZeichnen;
  SkylineBerechnen;
  SkylineAusgeben;
  SkylineZeichnen;
end.  {SkylineVonUtopolis}
```

2.2 Hast Du Töne?

2.2.1 Aufgabenstellung

Die zwölf Töne in der westlichenMusik sind wie folgt angeordnet:

C Cis|Des **D** Dis|Es **E** Eis|**F** Fis|Ges **G** Gis|As **A** Ais|B **H**

Die Bezeichnungen links und rechts der senkrechten Striche stellen dabei in der wohltemperierten Stimmung jeweils den gleichen Ton dar. Das Intervall zwischen je zwei Tönen dieser Zwölferreihe heißt Halbtonschritt (H); zwei Halbtonschritte ergeben einen Ganztonschritt (G). Eine Dur-Tonleiter, bestehend aus 8 Tönen, wird nun, ausgehend von einem bestimmten Grundton, nach dem Schema G-G-H-G-G-G-H gebildet, wobei beim Erreichen des Endes in der obigen Liste wieder an ihrem Anfang begonnen wird. Von den Noten mit doppeltem Namen enthält jede Tonleiter entweder nur solche, die links vom Strich stehen, oder nur solche rechts vom Strich. Außerdem kommen in keiner Tonleiter zwei Noten mit gleichem Anfangsbuchstaben vor (B entspricht Hes). Das Intervall von einem Ton in einer Tonleiter zum selben Ton heißt Prim, das Intervall von einem Ton zum nächsten Sekund, zum übernächsten Terz, dann Quart, Quint, Sext, Septim und Oktav.

Aufgabe:
Schreibe ein Programm, welches nach Eingabe eines Grundtons zunächst die Dur-Tonleiter dieses Grundtons ausgibt und anschließend Intervalle berechnet. Als Grundtöne braucht das Programm nur C, G, D, A, E, H, Des, As, Es, B und F zu akzeptieren.
Eingabe eines Punktes führt zur Aufforderung der Eingabe eines neuen Grundtons; wird hierfür auch ein Punkt eingegeben, so ist das Programm beendet.

Beispiel:
Grundton: A
A-Dur Tonleiter: A H Cis D E Fis Gis A
Intervall: Cis plus Quart
Cis plus Quart = Fis
Intervall: Es minus Terz
"Es" kommt in dieser Tonleiter nicht vor!
Intervall: .
Grundton: Es
Es-Dur Tonleiter: Es F G As B C D Es
Intervall: Es minus Terz
Es minus Terz = C
Intervall: .
Grundton: .

Sende drei Protokolle von Dialogen entsprechend dem Beispiel, darunter eines für die Eingabe:

D
Fis plus Terz
E minus Sext
Cis plus Quint
F plus Sekund

2.2.2 Lösungsidee

Um mit den Intervallen und den Ganz- und Halbtonschritten besser umgehen zu können, ordnen wir zunächst den Tönen Zahlen zu, beginnend mit C = 0, Des = 1, D = 2 usw. Dabei wählen wir bei zwei gleichhohen Tönen jeweils den Ton rechts vom Strich, da dieser als Grundton auftreten darf.

Nun müssen wir irgendwie dafür sorgen, daß wir für jede Tonleiter jeweils die richtigen Tonbezeichnungen erwischen, d. h. nur solche links vom Strich bzw. nur solche rechts davon. Dafür verwenden wir zwei Arrays *KreuzToene* und *BToene*. Beide enthalten die in der Aufgabenstellung fett gedruckten Töne C, D, E, F, G, A, H; *KreuzToene* enthält zusätzlich die Töne links vom Strich, d. h. die mit is-Endung, in *BToene* sind stattdessen die Töne rechts vom Strich, also As, B und die Töne mit es-Endung, enthalten.

Die erlaubten Grundtöne teilen wir in *KreuzTonarten* und *BTonarten* auf. Für Tonleitern mit einem Grundton aus den Kreuztonarten werden nur Töne aus *KreuzToene* benötigt; entsprechend werden für Tonleitern mit Grundton aus den B-Tonarten nur Töne aus *BToene* benötigt. Mit Grundkenntnissen der Musiktheorie ist die Aufteilung ganz einfach; aber auch ohne solche Kenntnisse läßt sich die Aufteilung leicht finden, indem man von jedem möglichen Grundton aus die Dur-Tonleiter nach dem angegebenen Schema von Ganz- und Halbtonschritten bildet. Dabei muß man natürlich darauf achten, daß eine Tonleiter keine Töne mit gleichem Anfangsbuchstaben enthält.

Wir erhalten als *KreuzTonarten* C, G, D, A, E, H und als *BTonarten* F, B, Es, As und Des. Das sind alle Dur-Tonarten mit bis zu fünf Vorzeichen. Nun definieren wir noch ein Array *Durtonleiter*, das mit den Intervallnamen indiziert wird und die Abstände der einzelnen Töne der Tonleiter vom Grundton in Halbtonschritten angibt, d. h. es enthält die Einträge 0, 2, 4, 5, 7, 9, 11 und 12. Sucht man nun beispielsweise zu einem beliebigen Grundton der Kreuztonarten die Septim, so findet man in *Durtonleiter*[*Septim*] den Wert 9 und rechnet nun in *KreuzToenen*: Grundton + 9 mod 12 und hat damit den gewünschten Ton gefunden. Die Modulo-Rechnung ist immer erforderlich, um nicht aus dem definierten Tonbereich herauszufallen.

Mit diesen Datenstrukturen können wir uns nun an die Lösung der vier Teilaufgaben geben:

1. Einlesen eines Grundtons und Überprüfung auf Korrektheit,
2. Ausgabe der entsprechenden Tonleiter,
3. Eingabe eines Tons und eines Intervalls und Überprüfung auf Korrektheit,
4. Ausgabe des neuen Tons.

Nach Eingabe eines Tons wird überprüft, ob er zu den zulässigen Grundtönen gehört. Alle erlaubten Grundtöne sind in der C-Dur- und der As-Dur-Tonleiter enthalten. Ob es sich um einen erlaubten Grundton handelt, kann man also prüfen, indem man die C-Dur- und die As-Dur-Tonleiter bildet und prüft, ob der eingegebene Ton in einer der beiden Tonleitern vorkommt. Natürlich kann man auch einfach den eingegebenen Ton mit der Menge der erlaubten Grundtöne vergleichen.

Für die Ausgabe der gewünschten Tonleiter durchläuft man das Durtonleiter-Array und zählt die Töne beginnend beim Grundton im Array *KreuzToene* oder *BToene* ab.

Beispiel:
Grundton sei As. Die gewünschte As-Dur-Tonleiter besteht aus den Tönen $8 =$ **As**, $8 + 2 =$ **B**, $8 + 4 \bmod 12 =$ **C**, $8 + 5 \bmod 12 =$ **Des**, $8 + 7 \bmod 12 =$ **Es**, $8 + 9 \bmod 12 =$ **F**, $8 + 11 \bmod 12 =$ **G** und $8 + 12 \bmod 12 =$ **As**.

Nach Eingabe eines Intervalls muß geprüft werden, ob der eingegebene Ton zur aktuellen Tonleiter gehört. Dazu kann man entweder die aktuelle Tonleiter zwischenspeichern und damit vergleichen oder man berechnet die Tonleiter neu, so wie dies auch für die Tonleiterausgabe geschieht. Diese Variante wurde im abgedruckten Programm gewählt. Außerdem wird geprüft, ob eine gültige Intervallbezeichnung und "plus" bzw. "minus" eingegeben wurde. Nur wenn die Eingabe völlig korrekt war, wird der zweite Ton des Intervalls ausgegeben. Dabei geht man vor wie bei der Beschreibungs des Arrays *Durtonleiter* bereits angegeben: Aus dem Intervallnamen und dem zugehörigen Wert im Array *Durtonleiter* ist ableitbar, wie viele Töne man in *KreuzToene* bzw. in *BToene* abzählen muß.

Die Lösung für das geforderte Beispiel lautet:
D E Fis G A H Cis D
Fis plus Terz = A
E minus Sext = G
Cis plus Quint = G
F plus Sekund: Ton in dieser Tonleiter nicht vorhanden !

2.2.3 Programmtext

```
Program Tonleitern;

uses crt;

const C = 0; Des = 1; D = 2; Es = 3; E = 4; F = 5;
      Ges = 6; G = 7; As = 8; A = 9; B = 10; H = 11;

      Prim = 0; Septim = 6; Oktav = 7;
      Fehler = -128; Abbruch = 127;

      KreuzToene : Array [C..H] of String =
      ('C', 'Cis', 'D', 'Dis', 'E', 'F', 'Fis', 'G', 'Gis', 'A', 'Ais', 'H');

      BToene : Array [C..H] of String =
      ('C', 'Des', 'D', 'Es', 'E', 'F', 'Ges', 'G', 'As', 'A', 'B', 'H');

      KreuzTonarten = [C, D, E, G, A, H];

      BTonarten = [Des, Es, F, As, B];

      IntervallNamen : Array [Prim..Oktav] of String =
         ('Prim', 'Sekund', 'Terz', 'Quart', 'Quint', 'Sext', 'Septim',
            'Oktav');

      Durtonleiter : Array [Prim..Oktav] of Shortint =
            (0, 2, 4, 5, 7, 9, 11, 12);

var Tonart : Shortint;
    Intervall : Shortint;
    Ausgangston : Shortint;

procedure TonartEingabe (var t : Shortint);
var Eingabe : String;
    Ton : Shortint;
begin
  t := Fehler;
  repeat
    write ('Grundton␣:␣');
    readln (Eingabe);
    for Ton := Prim to Oktav do
```

```
    begin
      if (Kreuztoene [(C + Durtonleiter [Ton]) mod 12]) = Eingabe then
        t := (C + Durtonleiter [Ton]) mod 12;
      if (BToene [(As + Durtonleiter [Ton]) mod 12]) = Eingabe then
        t := (As + Durtonleiter [Ton]) mod 12
    end;
    if Eingabe = '.' then
      t := Abbruch;
    if t = Fehler then
      writeln (#7, 'ungültige␣Eingabe␣!');
  until t <> Fehler
end;


procedure IntervallEingabe (t : Shortint; var TonZurueck, i : Shortint);
const Fehlermeldung : Array [1..3] of String =
                        ('Ton␣in␣dieser␣Tonleiter␣nicht␣vorhanden␣!',
                         'Intervall␣existiert␣nicht␣!',
                         'plus␣oder␣minus␣eingeben␣!');
var Eingabe : String;
    ETon, EInt, ESgn : String;
    FehlerNummer : Shortint;
    Zaehler : Shortint;
    Ton : Shortint;
begin
  i := Fehler;
  TonZurueck := Fehler;
  repeat
    FehlerNummer := 0;
    write ('Intervall␣eingeben␣(Grundton␣plus/minus␣Intervall):␣');
    readln (Eingabe);
    while Pos ('␣', Eingabe) = 1 do
      Delete (Eingabe, Pos ('␣', Eingabe), 1);
      (* Entfernt Leerzeichen vor Grundton des Intervalls *)
    ETon := copy (Eingabe, 1, Pos ('␣', Eingabe) -1);
    Eingabe := copy (Eingabe, Pos ('␣', Eingabe) + 1, Length (Eingabe));
    ESgn := copy (Eingabe, 1, Pos ('␣', Eingabe) - 1);
    EInt := copy (Eingabe, Pos ('␣', Eingabe) + 1, Length (Eingabe));
    for Zaehler := Prim to Oktav do
       if (IntervallNamen [Zaehler]) = EInt then
         i := Zaehler;
    if Eingabe = '.' then i := Abbruch;
```

```
    if ESgn = 'minus' then i := -i
    else if (ESgn <> 'plus') and (i <> Abbruch) then
      FehlerNummer := 3;
    for Ton := Prim to Septim do
        if ((t in KreuzTonarten) and
                (ETon = (KreuzToene [(Durtonleiter [Ton] + t) mod 12])))
            or ((t in BTonarten) and
                (ETon = (BToene [(Durtonleiter [Ton] + t) mod 12]))) then
            TonZurueck := Ton;
    if (TonZurueck = Fehler) and (i <> Abbruch) then
      FehlerNummer := 1;
    if i = Fehler then
      FehlerNummer := 2;
    if FehlerNummer <> 0 then
      writeln (#7, Fehlermeldung [Fehlernummer])
  until FehlerNummer = 0
end;

procedure TonleiterAusgabe (t : Shortint);
var Ton : Shortint;
begin
  write (BToene [t], '-Dur-Tonleiter␣:␣');
  for Ton := Prim to Oktav do
      if t in KreuzTonarten then
          write (KreuzToene [(t + Durtonleiter [Ton]) mod 12], '␣')
          else write (BToene [(t + Durtonleiter [Ton]) mod 12], '␣');
  writeln
end;

procedure IntervallAusgabe (t, Ton, i : Shortint);
begin
  if t in KreuzTonarten then
    write (KreuzToene [(Durtonleiter [Ton] + t) mod 12])
    else write (BToene [(Durtonleiter [Ton] + t) mod 12]);
  if i < 0 then write ('␣minus␣')
           else write ('␣plus␣');
  write (IntervallNamen [Abs (i)], '␣=␣');
  Ton := (Ton + Oktav + i) mod 7;
  if t in KreuzTonarten then
    writeln (KreuzToene [(Durtonleiter [Ton] + t) mod 12])
    else writeln (BToene [(Durtonleiter [Ton] + t) mod 12])
end;
```

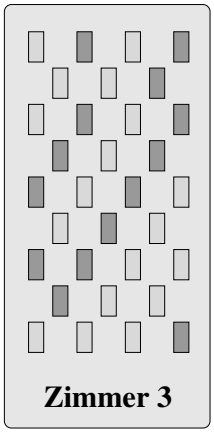
```
begin
  clrscr;
  TonartEingabe (Tonart);
  while Tonart <> Abbruch do
  begin
    TonleiterAusgabe (Tonart);
    IntervallEingabe (Tonart, Ausgangston, Intervall);
    while Intervall <> Abbruch do
    begin
      IntervallAusgabe (Tonart, Ausgangston, Intervall);
      IntervallEingabe (Tonart, Ausgangston, Intervall)
    end;
    TonartEingabe (Tonart)
  end;
  repeat until keypressed
end.
```

2.3 Hotelschlüssel

2.3.1 Aufgabenstellung

Wenn im Hotel "Graf Eberhard", benannt nach dem Gründer der Universität Tübingen, ein Gast eintrifft, wird für ihn ein neuer Schlüssel hergestellt. Dieser besteht aus einer Karte mit Löchern, die von einem Lichtstrahl im Türschloß abgetastet werden. Das Türschloß eines Zimmers wird jeweils passend eingestellt.

Beispiel:



Die Schlüssel müssen folgende Regeln erfüllen:

- Von den 32 möglichen Positionen sind wenigstens 5 mit einem Loch zu versehen. In unserem Beispiel sind es 14.
- Es darf keine der Spalten, Zeilen oder Diagonalen vollständig gelocht sein (damit der Schlüssel nicht so leicht bricht).
- Keine zwei Schlüssel dürfen identisch oder zueinander symmetrisch sein. Dadurch kann kein Schlüssel in einem anderen Zimmerschloß passen.

Aufgabe:
Schreibe ein Programm, das in der Lage ist, verschiedene Schlüssel zu entwerfen. Sende uns zehn verschiedene Entwürfe.

2.3.2 Lösungsidee

Vorab: Als Diagonale betrachten wir eine Folge von mindestens drei Löchern. Es gibt also auf jedem Schlüssel 12 Diagonalen.

Es sind zwei Entscheidungen zu treffen:

1. Welche Datenstruktur wird für die Hotelschlüssel verwendet?
2. Werden die Schlüssel zufällig oder systematisch gewählt?

Die Abbildung legt die Verwendung eines zweidimensionalen Arrays nahe. Dabei können entweder die Dimensionen 9×4 oder 9×7 gewählt werden. Bei 9×4 entspricht dem letzten Feld in den Zeilen 2, 4, 6 und 8 keine Lochposition auf dem Schlüssel; bei 9×7 gibt es in jeder Zeile mehrere Einträge (abwechselnd drei und vier), für die keine Lochposition existiert.

Man kann den Schlüssel aber auch als eindimensionales Array der Größe 32 speichern, dann entspricht jedem Feld genau eine Lochposition.

Die Prüfung, ob eine Zeile, Spalte oder Diagonale vollständig gelocht wurde, ist bei Verwendung eines 9×7-Arrays am einfachsten, allerdings muß man berücksichtigen, daß man die Felder des Arrays, die keiner Lochposition entsprechen, übergeht. Bei anderen Strukturen, insbesondere dem eindimensionalen Array, ist die sicherste Lösung, die Felder, die eine Zeile, Spalte oder Diagonale bilden, als Konstanten zu definieren. Dann läßt sich jeweils leicht überprüfen, ob alle Felder, die zu einer Zeile (Spalte, Diagonale) gehören, im neuen Schlüssel gelocht wurden.

Nun zur Frage der Schlüsselwahl. Sollen die Schlüssel zufällig erzeugt werden, bestimmt man zunächst eine Zahl zwischen 5 und 23. Das genügt als obere Grenze, denn bei allen Schlüsseln mit mehr als 23 Löchern muß mindestens eine Zeile vollständig gelocht sein.

Anschließend wählt man für jedes Loch zufällig eine Position. Wie dies im Detail geschieht, hängt natürlich wieder von der Datenstruktur ab. Beim eindimensionalen Array muß man lediglich eine Zahl zwischen 1 und 32 wählen. Trifft man mit der neuen Position auf ein schon gewähltes Loch, hat man wiederum mehrere Möglichkeiten:

- Verringerung der zu lochenden Positionen (nur möglich, wenn mehr als 5 Positionen gelocht werden sollten);
- Verschiebung des Loches nach rechts, links, oben oder unten, je nachdem, was möglich ist;
- neue Wahl einer Position.

Nachdem die gewünschte Anzahl Löcher berechnet wurde, muß der Schlüssel auf Bruchfestigkeit und Symmetrie bzw. Gleichheit mit bereits erzeugten Schlüsseln überprüft werden. Die Prüfung auf Bruchfestigkeit wurde bereits erwähnt. Bei der Symmetrieprüfung genügt es, nur die vertikale Symmetrie zu betrachten, da der Schlüssel wegen des längeren Griffstücks nicht um die horizontale Achse gedreht werden kann. Schließlich muß noch überprüft werden, ob dieser Schlüssel bereits erzeugt wurde. Das bedeutet, daß man alle Schlüssel, die während des Programmlaufs erzeugt werden, speichern muß.

Man kann die Schlüssel auch systematisch erzeugen, etwa, indem man zunächst alle Schlüssel mit 5 Löchern, dann alle mit 6 Löchern usw. erzeugt. Verwendet man das eindimensionale Array, so kann man dieses als 32-stellige Binärzahl auffassen, die hochgezählt wird. Hat beispielsweise die Binärdarstellung eines Schlüssels den Wert 123, so wird als nächstes der Schlüssel mit dem Wert 124 erzeugt. Hierbei werden die Schlüssel natürlich nicht in der Reihenfolge wachsender Löcherzahlen erzeugt.

Bei systematischer Erzeugung entfällt die Prüfung auf Gleichheit mit bereits erzeugten Schlüsseln. Die Symmetrie kann geprüft werden, indem man den symmetrischen Schlüssel erzeugt und überprüft, ob er gemäß der Reihenfolge, in der die Schlüssel erzeugt werden, bereits hätte erzeugt werden müssen.

2.3.3 Halbformale Programmbeschreibung

Im abgedruckten Programm werden alle Schlüssel der Reihe nach erzeugt. Man kann wählen, ob man eine bereits erzeugte Schlüsselserie fortsetzen möchte oder ob man wieder ganz von vorne, d.h. mit dem Schlüssel mit kleinstem Wert, beginnen möchte. Da dies in der Aufgabenstellung nicht verlangt war, geben wir es in der halbformalen Programmbeschreibung nicht mit an.

```
Eingabe der Anzahl gewünschter Schlüssel;
Wiederhole:
  berechne nächsten Schlüssel (aktueller Schlüssel)
bis gewünschte Anzahl Schlüssel erreicht
      oder alle möglichen Schlüssel erzeugt.

berechne nächsten Schlüssel (aktueller Schlüssel):
  erhöhe aktuellen Schlüssel um 1 = neuer Schlüssel;
  prüfe neuen Schlüssel auf Gültigkeit:
    Anzahl Löcher ≥ 5?
    Keine Spalte gelocht?
    Keine Zeile gelocht?
```

```
    Keine Diagonale gelocht?
    Nicht symmetrisch zu bereits erzeugtem Schlüssel?
  wenn gültig:
    Ausgabe des neuen Schlüssels
  sonst:
    berechne nächsten Schlüssel (neuer Schlüssel);
```

2.3.4 Programmtext

```
program Hotelschluessel;

uses crt;

const Zeile: array[1..9,1..4] of byte =          { In diesen Konstanten werden }
        ((1,10,19,28),(2,11,20,29),              { den Zeilen, Spalten und     }
         (3,12,21,30),(4,13,22,31),              { Diagonalen ihre zugehörigen }
         (5,14,23,32),(6,15,24,0),               { Lochnummern zugeordnet.     }
         (7,16,25,0),(8,17,26,0),                { Eine '0' dient hierbei nur  }
         (9,18,27,0));                           { als Platzhalter und hat     }
      Spalte: array[1..7,1..5] of byte =         { keine Bedeutung.            }
        ((1,2,3,4,5),(6,7,8,9,0),
         (10,11,12,13,14),(15,16,17,18,0),
         (19,20,21,22,23),(24,25,26,27,0),
         (28,29,30,31,32));
      Diagonale: array[1..12,1..7] of byte =
        ((1,6,11,16,21,26,31),(2,7,12,17,22,27,32),
         (4,8,12,16,20,24,28),(5,9,13,17,21,25,29),
         (10,15,20,25,30,0,0),(3,8,13,18,23,0,0),
         (14,18,22,26,30,0,0),(3,7,11,15,19,0,0),
         (2,6,10,0,0,0,0),(19,24,29,0,0,0,0),
         (23,27,31,0,0,0,0),(4,9,14,0,0,0,0));

type keytyp = array[0..33] of boolean;
                    { Typendeklaration für den Schlüssel   }
                    { In diesem Array finden sich die 32   }
                    { Löcher eines Schlüssels wieder.      }
                    { 0 und 33 dienen nur dem Schutz       }
                    { gegen Bereichsüberschreitung.        }
                    { (Der englische Bezeichner key ist bei }
                    { ansonsten deutschen Namen unschön,   }
                    { aber so praktisch kurz.)             }
```

```
var i,anzahl :integer;
    a,b,c,d,
    letzteKombination :boolean;
    key :keytyp;                                    { Globale Variablen }
    eingabe :string;

function bin2hex(a,b,c,d: boolean): char;
      { Wandelt eine 4-Bit Binärzahl in ein Hexadezimalzeichen um }
      { a,b,c,d repräsentieren die 4 Bit }
var zwi: char;
begin
  zwi := '0';
  if d then inc(zwi,8);
  if c then inc(zwi,4);
  if b then inc(zwi,2);
  if a then inc(zwi,1);
  if ord(zwi) > 57 then inc(zwi,7);      { Oberhalb der Ziffer 9, dann A-F }
  bin2hex := zwi
end;

procedure hex2bin(ahex: char; var a,b,c,d: boolean);
   { Wandelt ein Hexadezimal-Zeichen in einen 4-Bit Wert um. }
var i : integer;
begin
  ahex := Upcase(ahex);                          { Erstmal große Zeichen }
  if ord(ahex) > 57 then dec(ahex,55)     { A-F ? Ja, dann -55 sonst -48 }
  else dec(ahex,48);
  i := ord(ahex);
  if i >= 8 then begin
    d := true;
    dec(i,8)
    end
  else d := False;
  if i >= 4 then begin
    c := true;
    dec(i,4)
    end
  else c := false;
  if i >= 2 then begin
    b := true;
    dec(i,2)
    end
```

```
  else b := false;
  if i >= 1 then begin
    a := true;
    dec(i,1)
    end
  else a := false;
end;

procedure increase(var key: keytyp; var overflow: boolean);
  { Erhöht einen 32-Bit Wert um 1 }
  { Bei Überschreitung des Wertebereichs wird das Overflow-Flag gesetzt }
var i,c: integer;
begin
  c := 1;
  while (c < 33) and key[c] do inc(c);
      { Array bis '33', sonst Bereichsüb. }
  if c = 33 then overflow := true;
  key[c] := true;
  for i := c-1 downto 1 do key[i] := false
end;

function key2code(key: keytyp): string;
 { Wandelt einen Schlüssel (32-Bit Wert) in seinen Hexadezimalcode um, }
 { und gibt diesen als String zurück }
var code: string;
    i : integer;
begin
  for i := 0 to 7 do
  code[8-i] := bin2hex(key[i*4+1],key[i*4+2],key[i*4+3],key[i*4+4]);
  code[0] := chr(8);
  key2code := code
end;

procedure code2key(code: string; var key: keytyp);
    { Wandelt einen Hexadezimalcode (String) in einen }
    { 32-Bit Wert um (Keytyp) }
begin
  for i := 1 to 8 do
  hex2bin(code[9-i],key[i*4-3],key[i*4-2],key[i*4-1],key[i*4]);
end;
```

```
function AnzahlLoecher(key: keytyp): integer;
    { Zählt die Anzahl der gelochten Stellen auf dem Schlüssel }
var i,count: integer;
begin
   count := 0;
   for i := 1 to 32 do if key[i] then inc(count);
   AnzahlLoecher := count
end;

function SpalteGelocht(key: keytyp): boolean;
   { Gibt, falls eine Spalte vollständig gelocht ist, true zurück }
var x,y: integer;
    EineSpalte: boolean;
begin
  SpalteGelocht := false;
  for x := 1 to 7 do                                        { 7 Spalten }
     begin
       EineSpalte := true;                          { Aktuelle Spalte auf true }
       for y := 1 to 5 do
         if Spalte[x,y] > 0 then
             if not key[Spalte[x,y]] then
                EineSpalte := false;                          { doch nicht ? }
       if EineSpalte then
         SpalteGelocht := true                     { Aktuelle Spalte Gelocht ? }
     end
end;

function ZeileGelocht(key: keytyp): boolean;
   { Gibt, falls eine Zeile vollständig gelocht ist, true zurück. }
var x,y : integer;
    EineZeile : boolean;
begin
  ZeileGelocht := false;
  for x := 1 to 9 do
    begin
      EineZeile := true;
      for y := 1 to 4 do
        if Zeile[x,y] > 0 then
           if not key[Zeile[x,y]] then EineZeile := false;
      if EineZeile then ZeileGelocht := true;
    end
end;
```

```
function DiagonaleGelocht(Key : keytyp) : boolean;
   { Gibt, falls eine Diagonale vollständig gelocht ist, true zurück. }
var x,y : integer;
    EineDiag : boolean;
begin
DiagonaleGelocht := false;
for x := 1 to 12 do
  begin
    EineDiag := true;
    for y := 1 to 7 do
      if Diagonale[x,y] > 0 then
        if not key[Diagonale[x,y]] then EineDiag := false;
    if EineDiag then DiagonaleGelocht := true;
  end
end;

function KarteSchonDa (a,b : keytyp) : boolean;
 { Vergleicht zwei Karten nach dem Binärwert.                    }
 { Falls a < b ist, bedeutet dies, daß die Karte a in der        }
 { laufenden Schlüsselserie schon einmal erzeugt worden ist.     }
 { (Verwendet in der Prüfung auf Symmetrie.)                     }
var i : integer;
begin
  i := 32;
  while ( i > 0 ) and ( a[i] = b[i]) do dec(i);
  if a[i] < b[i] then KarteSchonDa := true
    else KarteSchonDa := false
end;

function symmetrisch(key : keytyp) : boolean;
   { Gibt true zurück, falls Key zu einer bereits erzeugten Karte }
   { achsensymmetrisch ist. }
var i : integer;
    symKey : keytyp;
begin
  for i := 1 to 5 do                  { Aus Key einen achsensymmetrischen }
    begin                               { (zur y-Achse) Schlüssel erzeugen. }
      symKey[i] := key[27+i];
      symKey[9+i] := key[18+i];
      symKey[18+i] := key[9+i];
      symKey[27+i] := key[i];
    end;
```

```
  for i:= 1 to 4 do
    begin
      symKey[5+i] := key[23+i];
      symKey[23+i] := key[5+i];
      symKey[14+i] := key[14+i];
    end;
    symmetrisch := KarteSchonDa(symKey,key);
    {Karte bereits erzeugt worden ? }
end;

function gueltig(key:keytyp) : boolean;
  { Zusammenfassung aller Einzel-Regeln. Gibt true zurück, wenn der }
  { Schlüssel gültig ist. }
begin
  gueltig := false;
  if ( AnzahlLoecher(Key) > 4 ) then
    if ( not SpalteGelocht(Key) ) then
      if ( not ZeileGelocht(Key) ) then
        if ( not DiagonaleGelocht(Key) ) then
          if ( not Symmetrisch(Key) ) then
            gueltig := true
end;


{ ———————— Ausgaberoutine ————————————— }

procedure GibSchluesselAus (key: keytyp; keynr : integer);
{ Gibt einen Schlüssel mit dessen laufender Nummer sowie dessen  }
{ Kombination in Hexadezimalschreibweise auf dem Bildschirm aus. }
var x, y : integer;
    eingabe : string[10];
begin
  clrscr;
  for x := 0 to 3 do
    for y := 1 to 5 do
    begin
      gotoxy(x*2+12,y*2);
      if key[x*9+y] then write ('O')          { 'O': gelocht, 'X': ungelocht }
                    else write ('X')
    end;
  for x := 0 to 2 do
    for y := 1 to 4 do
```

```
    begin
    gotoxy(x*2+13,y*2+1);
      if key[x*9+y+5] then write ('O')
                      else write ('X')
    end;
  gotoxy(1,12);
  writeln(' Schlüsselnummer: ',keynr);
  writeln;
  writeln(' Schlüsselcode: ',key2code(key));
  writeln;
  write('<RETURN>');
  readln
end;

{ ------- Hauptprogramm ---------- }
begin
  letzteKombination := false;
  clrscr;
  writeln('12. Bundeswettbewerb Informatik');
  writeln('3. Aufgabe ``Hotelschlüssel''');
  writeln;
  writeln;
  writeln('Möchten Sie eine neue Serie von Schlüsseln erstellen,');
  writeln('oder eine vorhandene Serie ergänzen?');
  write('(`N`eu / `E`rgänzen): ');
  repeat
    gotoxy(23,6);
    clreol;
    readln(eingabe)
  until upcase(eingabe[1]) in ['N','E'];
  if upcase(eingabe[1]) = 'N' then
    code2key('00000000',key)
  else
  begin
    writeln;
    write('Bitte Code des letzten Schlüssels der alten
        Serie eingeben: ');
    repeat
      gotoxy(61,8);
      clreol;
      readln(eingabe)
    until length(eingabe) = 8;
```

```
    code2key(eingabe,key)
  end;
  writeln;
  write('Bitte␣nun␣Anzahl␣der␣zu␣erzeugenden␣Schlüssel␣eingeben:␣');
  readln(anzahl);
  i := 1;
  repeat
    increase(key,letzteKombination);
    if gueltig(key) then
    begin
      GibSchluesselAus(key,i);
      inc(i)
    end
  until (i = anzahl + 1) or letzteKombination;   {bis alle Schlüssel erzeugt }
  if letzteKombination then
  begin
    writeln('Letzte␣Schlüsselkombination␣erreicht!');
    writeln('Es␣können␣keine␣weiteren');
    writeln('Schlüssel␣mehr␣erzeugt␣werden...')
  end;
  writeln;
  writeln('Programm␣ist␣hiermit␣beendet.');
  writeln('Falls␣diese␣Schlüsselserie␣zu␣einem');
  writeln('späteren␣Zeitpunkt␣noch␣ergänzt␣werden␣soll,');
  writeln('notieren␣Sie␣sich␣unbedingt␣den␣Schlüsselcode␣des␣');
  writeln('letzten␣Schlüssels␣und␣geben␣Sie␣ihn␣dann␣später␣');
  writeln('unter␣dem␣Menüpunkt␣`Ergänzen`␣wieder␣ein.');
  writeln;
  write('<RETURN>');
  readln
end.
```

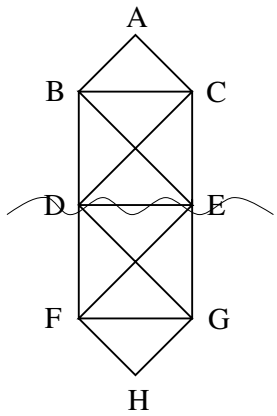
2.4 Das Haus des Nikolaus

2.4.1 Aufgabenstellung

Für das Zeichnen des Haus des Nikolaus gelten die bekannten Regeln:

- Der Streckenzug muß ohne Absetzen gezeichnet werden.
- Jede Strecke wird genau einmal durchlaufen.

Nun steht das Haus des Nikolaus am Ufer eines Sees, in dem es gespiegelt wird. Es entsteht das folgende Bild:



Aufgabe:
Schreibe ein Programm, das alle möglichen Wege ausgibt, die nach den oben angegebenen Regeln dieses Bild entstehen lassen! Gib dabei die Reihenfolge der durchlaufenen Punkte mit der oben angegebenen Bezeichnung aus!

2.4.2 Lösungsidee

Alle möglichen Wege müssen am Punkt D beginnen und an E enden bzw. umgekehrt an E beginnen und an D enden. Dies liegt daran, daß von D und E jeweils 5 Kanten ausgehen, also eine ungerade Anzahl, während von allen anderen Punkten eine gerade Anzahl Kanten ausgeht. Bei gerader Anzahl kann man einen Punkt jeweils über eine Kante erreichen und über eine andere wieder verlassen; die Punkte mit ungerader Anzahl Kanten muß man dagegen einmal öfter verlassen als erreichen (also Anfangspunkt eines Weges) oder einmal öfter

erreichen als verlassen (Endpunkt). Außerdem ergibt sich jeder Weg von E nach D als Umkehrung eines Weges von D nach E, so daß es genügt, nur D als Startpunkt zu betrachten und jeden gefundenen Weg auch in umgekehrter Richtung auszugeben.

Zum Finden der Wege bietet sich eine rekursive Vorgehensweise an. Ausgehend von D gelangt man über eine noch unbenutzte Kante zu einem Punkt, der nun als zweiter Punkt auf dem Weg Ausgangspunkt für die weitere Suche nach einer unbenutzten Kante wird. Wird die Suchprozedur für einen Punkt an Position 16 auf dem Weg aufgerufen, wurde eine Lösung gefunden, die ausgegeben werden kann (und ihre Umkehrung ebenfalls).

2.4.3 Programm-Dokumentation

Im Array *kante* werden die Bezeichnungen der 15 Kanten als Konstanten gespeichert. Welche dieser Kanten auf dem aktuellen Weg bereits benutzt wurden, gibt das Array *benutzt* an. Die Abfolge der Punkte auf dem aktuellen Weg wird im Array *folge* gespeichert.

2.4.4 Halbformale Programmbeschreibung

```
Initialisiere notwendige Variable;
Durchlaufe Figur von Punkt D aus,
    wobei D an erster Position auf dem Weg auftritt;
gib Gesamtzahl der Lösungen aus.

Durchlaufe Figur von Punkt X an Position k:
  wenn k = 16:
    gib Folge der Punkte in folge aus;
    gib Folge der Punkte in folge in umgekehrter Reihenfolge aus;
  sonst:
    betrachte alle 15 Kanten der Reihe nach:
      wenn Kante noch nicht benutzt:
        wenn Anfangs- oder Endpunkt der Kante = X:
          durchlaufe vom anderen Punkt der Kante an Position k+1;
```

Bildschirmausgabe:
1. Lösung: DBACBECDEFDGFHGE 2. Lösung: EGHFGDFEDCEBCABD
3. Lösung: DBACBECDEFDGHFGE 4. Lösung: EGFHGDFEDCEBCABD
...
13757. Lösung: DGHFGEFDECDBACBE 13758. Lösung: EBCABDCEDFEGFHGD

13759. Lösung: DGHFGEFDECDBCABE 13760. Lösung: EBACBDCEDFEGFHGD
Anzahl der Lösungen: 13760

2.4.5 Programmtext

```
program Nikolaus;

const kante : array[1..15] of string[2] =
    ('AB','AC','BC','BD','BE','CD','CE','DE',
     'DF','DG','EF','EG','FG','FH','GH');

var benutzt : array[1..15] of boolean;
     folge : array[1..16] of char;
     anzloesungen, i : integer;

procedure durchlaufe(punkt: char; pos: integer);
{pos: Position des Punktes beim Durchlaufen der Folge}
var i : integer;

begin
  folge[pos] := punkt;
  if pos = 16 then
  begin
     write(anzloesungen+1,'.te␣Lösung:␣');
     for i := 1 to 16 do
        write(folge[i]);
     write('␣␣␣');
     write(anzloesungen+2,'.te␣Lösung:␣');
     for i := 16 downto 1 do
        write(folge[i]);
     writeln;
     anzloesungen := anzloesungen + 2;
  end
  else
     for i := 1 to 15 do
        if not benutzt[i] then
           if kante[i,1] = punkt then
           begin
              benutzt[i] := true;
              durchlaufe(kante[i,2],pos+1);
              benutzt[i] := false
           end
```

```
        else
          if kante[i,2] = punkt then
          begin
            benutzt[i] := true;
            durchlaufe(kante[i,1],pos+1);
            benutzt[i] := false
          end
end;

begin
  anzloesungen := 0;
  for i := 1 to 15 do
    benutzt[i] := false;
  durchlaufe('D',1);
  writeln('Anzahl␣der␣Lösungen:␣',anzloesungen);
  readln
end.
```

2.4.6 Lösung in Haskell

Wir beschreiben nun noch, wie die Aufgabe mit der funktionalen Programmiersprache Haskell[1] gelöst werden kann.[2] Es handelt sich dabei aber nicht um eine Einführung in Haskell für absolute Neulinge, sondern einige Grundkenntnisse funktionaler Programmiersprachen werden vorausgesetzt.

Reine funktionale Programmierung unterscheidet sich grundlegend von imperativ orientierter Programmierung, wie sie beispielsweise von Sprachen wie C oder Java her bekannt ist. Ein funktionales Programm stellt gewissermaßen eine *ausführbare Spezifikation* eines Problems dar.

Es gibt für das Nikolaus-Problem eine Reihe grundsätzlich verschiedener Lösungswege, doch drängt sich hier ein Ansatz besonders auf: Die Zugfolgen lassen sich in einem Baum darstellen. Es ist nicht ungewöhnlich, in funktionalen Sprachen bei ähnlichen Problemen zunächst eine geeignete Definition für den kompletten Zugbaum zu suchen, und im folgenden nur auf diesem Baum zu arbeiten. Eine wesentliche Besonderheit von Haskell kommt uns hier entgegen: Grundsätzlich werden nur solche Zwischenergebnisse berechnet, die für die Berechnung des Endresultats auch wirklich notwendig sind, zudem werden unnötige Doppelberechnungen desselben Werts vermieden. So kann ein Haskell-Programm mit sehr großen oder sogar unendlichen Bäumen umgehen; nur die Teile, welche wirk-

[1] siehe z. B. `http://www.haskell.org`

[2] Diese Lösung wurde von Thomas Fischbacher ausgearbeitet.

lich auch benötigt werden, werden ausgerechnet (und vergessen, sobald sie nicht mehr gebraucht werden). Das ist ein entscheidender Unterschied zu imperativer Programmierung; dort muß der Programmierer selbst dafür sorgen, Zwischenergebnisse nicht ständig neu zu berechnen.

Der gesamte Lösungsbaum soll nur zulässige Wege enthalten; in jedem Knoten sollen Information über die zuletzt erreichte Ecke sowie alle Möglichkeiten, die Zugfolge fortzusetzen, gespeichert sein. Endknoten spielen eine Sonderrolle.

Die Forderung, daß der Zugbaum nur zulässige Wegzüge enthalten soll und nicht etwa abgebrochene Fehlversuche, läßt sich am einfachsten durch folgende Zugregel erfüllen: Von einer gegebenen Ecke aus darf nur über solche unbenutzte Kanten weitergelaufen werden, für welche sich letztendlich ein erlaubter Streckenzug ergibt. Diese Zugregel sieht zunächst sehr sonderbar aus, da sie offenbar nichtlokal ist. Allerdings ist die Umsetzung für die Fälle "null Kanten" und "eine Kante" trivial, und es ist einfach, das Problem für n Kanten zu lösen, wenn es bereits für $n-1$ Kanten gelöst wurde. Somit läßt sich der Zugbaum für eine gegebene Startecke definieren, und wir müssen uns lediglich noch mit der Frage der möglichen Startecken und der Übersetzung der im Zugbaum enthaltenen Lösungen in Zeichenketten beschäftigen.

Zunächst definieren wir die Datenstruktur für den Zugbaum:

```
data Zbaum a = Fehlschlag | Ende a | Zuege (a, [Zbaum a])
```

Mit `data` werden in Haskell neue Datentypen definiert. Namen von Datentypen (und Konstruktoren) werden grundsätzlich groß geschrieben, Namen von Werten (Variablen, Funktionen) stets klein. Der Datentyp `Zbaum` besitzt drei sogenannte "Konstruktoren". Wir müssen uns nicht festlegen, welchen Datentyp wir für die Ecken verwenden (denkbar wären etwa Buchstaben, Zeichenketten, Zahlen oder ein eigener Datentyp `Ecke`), und hier ist das auch nicht geschehen. Dies besagt der Typ-Parameter `a`. Ein Objekt vom Typ `Zbaum Char` wäre ein Zugbaum, in dem die Ecken durch Buchstaben beschrieben werden, ein Zugbaum, in dem die Ecken durch Zahlen beschrieben werden, hätte etwa den Typ `Zbaum Int`.

Ein `Zbaum a` kann entweder ein `Fehlschlag` sein (wenn es keinen erlaubten Streckenzug gibt) oder eine Ansammlung mehrerer `Zuege`, dargestellt durch ein geordnetes Paar aus einer Startecke und einer Liste von Fortsetzungen, welche jeweils wiederum ein `Zbaum a` sind. Weil sich ein `Zbaum a` nicht unendlich in die Tiefe erstreckt und am Ende keine Fehlschläge sitzen dürfen, gibt es den dritten Konstruktor `Ende`. Ein `Ende a` ist eine End-Ecke.

Kommen wir zum interessantesten Teil des Programms. Wir benötigen eine Funktion, welche zu einer gegebenen Startecke und Kantenmenge den gesamten Zugbaum berechnet.

Für den Fall, daß es keine oder nur eine Kante gibt, ist das Problem leicht zu lösen. Im Falle mehrerer Kanten ist eine einfache Vorgehensweise, sich der Reihe nach alle Kanten anzusehen, für jede Kante zu prüfen, ob sie eine mögliche Fortsetzung des Weges darstellt, und die möglichen Fortsetzungen in einer Liste zu sammeln, welche dann in den Baum eingebaut wird.

Es ist zweckmäßig, vier Parameter zu verwenden: Die Startecke `se` des Zugbaums, eine Liste `rk` der noch nicht überprüften unbenutzten Kanten, eine Liste `bk` der bereits betrachteten unbenutzten Kanten, und eine Liste `bf` der bereits berechneten möglichen Fortsetzungen.

Falls wir uns alle unbenutzten Kanten angesehen haben, und keine mögliche Fortsetzung gefunden haben, ist der Zugbaum ein `Fehlschlag`. Falls es aber doch Fortsetzungen gab, ist der Zugbaum ein Knoten, in dem die Startecke und die Liste der Fortsetzungen enthalten sind.

Falls es noch genau eine nicht betrachtete unbenutzte Kante gibt, kommt es darauf an, ob diese Kante von der Startecke wegführt. Falls nicht, ist der Zugbaum ein `Fehlschlag`, ansonsten ein Knoten, in dem die Startecke und eine einzige Fortsetzung — ein `Ende` — eingetragen sind.

Ansonsten, d.h. wenn es mehr als eine noch nicht betrachtete Kante gibt, sehen wir uns die erste solche Kante an; wenn sie nicht von unserer Startecke wegführt, übergehen wir sie; das Ergebnis ist in diesem Fall dasjenige, das wir erhalten, wenn wir besagte Kante zu den bereits betrachteten unbenutzten Kanten hinzunehmen und den Zugbaum ausrechnen. Falls aber die Kante von unserer Startecke wegführt, sehen wir uns zunächst an, ob wir den Weg vom anderen Ende aus fertigzeichnen können. Hierfür berechnen wir den Zugbaum, dessen Startecke das ferne Ende unserer Kante ist, wobei die Menge der noch unbenutzten Kanten hier die Gesamtmenge der (betrachteten und unbetrachteten) unbenutzten Kanten *ohne* die Kante ist, welche uns zum potentiellen neuen Startpunkt gebracht hat. Wenn dieser "kleinere Zugbaum" ein Fehlschlag ist, können wir die Kante nicht benutzen. Wir berechnen dann den Zugbaum für unsere momentane Situation, mit der Änderung, daß wir die Kante, für die wir keine Fortsetzung gefunden haben, zu den betrachteten hinzunehmen. Falls es vom anderen Ende der Kante aber weitergeht, machen wir dasselbe, nur nehmen wir den gefundenen Zugbaum zur Liste der berechneten Fortsetzungen hinzu. In natürlicher Sprache mag dies alles ein wenig verwirrend klingen, deswegen folgt eine äquivalente Spezifikation, welche ganz zufällig auch ein ausführbares Haskell-Programm ist. (Um weiter unten Bezug auf Teile dieser Definition nehmen zu können, sind die Zeilen numeriert. Die Zeilennummern gehören nicht zum Programm.)

```
1   zb _  []              _   [] = Fehlschlag
2   zb se []              _   bf = Zuege (se, bf)
3   zb se [(a,b)]         []  bf
4    | se == a   = Zuege (se, [Ende b]) -- bf kann hier
5    | se == b   = Zuege (se, [Ende a]) -- nur leer sein!
6    | otherwise = Fehlschlag
7   zb se ((k@(a,b)):rk) bk  bf
8    | se == a   = weiter b
9    | se == b   = weiter a
10   | otherwise = zb se rk (k:bk) bf
11   where weiter ne =
12           let kind = zb ne (rk ++ bk) [] []
13           in case kind of Fehlschlag  -> zb se rk (k:bk) bf
14                           x           -> zb se rk (k:bk) (x:bf)
```

Für unsere erste Haskell-Funktionsdefinition ist dies ein Sprung ins kalte Wasser; so ziemlich alle in Haskell häufiger verwendeten Konstrukte kommen hierin vor. Dieses Dickicht soll nun ein wenig entwirrt werden:

- Wenn das Haskell-System eine Funktion ausrechnet, werden von oben nach unten alle Definitionen dieser Funktion durchprobiert, bis eine gefunden wird, für die die auf der linken Seite des Gleichheitszeichens stehenden Muster auf die Parameter (Eingabewerte der Funktion) passen. In diesem Fall wird die rechte Seite ausgerechnet, wobei hier auftretende Variablen die Werte aus ihren Bindungen an die Muster auf der linken Seite erhalten. Der Unterstrich ist ein "Joker".

- Kommentare werden durch zwei Minuszeichen eingeleitet und laufen bis zum Ende der Zeile.

- Muster können nur die generelle Struktur von Ausdrücken prüfen; kompliziertere Dinge wie etwa die Überprüfung, ob ein Teil eines Parameters mit einem anderen identisch ist, können nur auf der rechten Seite des Gleichheitszeichens erledigt werden, etwa mit `if/then/else` oder aber eleganter mit einer Fallunterscheidung; der senkrechte Strich gibt die einzelnen Fälle an, welche wiederum von oben nach unten durchgesehen werden.

- Das Muster `(k@(a,b)):rk` paßt auf eine (nichtleere) Liste, deren Kopf auf das Muster `k@(a,b)` paßt, und deren Schwanz als `rk` auf der rechten Seite des Gleichheitszeichens verfügbar ist. Das Muster `k@(a,b)` paßt auf ein Paar von Werten, die einzeln als `a`, `b`, zusätzlich aber auch als Paar als `k` auf der rechten Seite des Gleichheitszeichens bekannt sind.

- Das `where`-Konstrukt führt eine lokale Definition ein — hier der Funktion `weiter`, deren Parameter mit `ne` ('nächste Ecke') bezeichnet wird; `where` arbeitet im nachhinein — die hier definierte Funktion wurde vor der Definition bereits benutzt, und klammert zudem über die verschiedenen Fälle der Fallunterscheidung. Nur am Rande sei bemerkt, daß die Einrückung des Programmtexts in gewissem Umfang auch Auswirkungen darauf hat, wie er vom Haskell-System aufgefaßt wird.
- Mit `let/in` lassen sich lokal Werte definieren, etwa im Sinne eines mathematischen "sei im folgenden ...".
- Während '`:`' eine Liste um ein Kopfelement erweitert, hängt '`++`' zwei Listen zusammen.
- `case/of` liefert eine Fallunterscheidung anhand der Struktur eines Ausdrucks genau wie wir es bereits von Funktionsdefinitionen her kennen. Auch die Mustererkennung funktioniert genauso.

Beinahe der gesamte interessante Code für das Nikolaus-Problem steckt in den 14 Zeilen der Definition von `zb`. Haskell besitzt ein starkes Typsystem, das viele triviale Programmierfehler entdeckt, weshalb auch Anfänger schon nach kurzer Zeit relativ komplizierte Funktionen definieren können.

Nun müssen nur noch die im Baum enthaltenen Zugfolgen zur Ausgabe in eine Liste von Zeichenketten umgerechnet werden. In Haskell sind Zeichenketten Listen von Zeichen; bisher haben wir uns nicht festgelegt, welchen Datentyp wir für die Ecken verwenden. Der Zeichen-Datentyp `Char` bietet sich an, denn eine Liste der nacheinander durchlaufenen Ecken ist somit gleichzeitig auch eine Zeichenkette.

Wir definieren eine Funktion `wegeInBaum`, welche zu einem vorgegebenen Zugbaum eine Liste aller enthaltenen Wege (in Form von Listen von Ecken) berechnet.

Für einen `Fehlschlag` ist die Liste von Wegen leer. Für ein `Ende x` besteht sie aus einem einzigen Weg, und dieser enthält nur die Ecke `x`. Für den allgemeinen Knoten `Zuege (ek,kbe)`[3] erhalten wir alle Wege, indem wir die Ecke `ek` an den Anfang aller Wege stellen, die in den Unterbäumen `kbe` enthalten sind. Hier die Haskell-Definition:

```
wegeInBaum Fehlschlag = []
wegeInBaum (Ende x)   = [[x]]
wegeInBaum (Zuege (ek, kbe)) =
 [(ek:wkb) | k <- kbe, wkb <- wegeInBaum k]
```

[3] `ek` soll für "Ecke", `kbe` für "kleinere Bäume" stehen

Die Schreibweise `[f x | x <- y]` ist an die mathematische Mengenschreibweise angelehnt. Obiges liest sich als "Liste aller Ergebnisse der Anwendung von `f` auf `x`, wobei `x` aus der Liste `y`" stammt. Die rechte Seite der letzten Zeile der Definition liest sich: "Liste aller am Anfang um `ek` erweiterten Wege `wkb`[4], wobei `k` aus der Liste der kleinen Bäume `kbe` stammt, und `wkb` alle Wege in `k` sind."

Wir hatten bereits in Abschnitt 2.4.2 festgestellt, daß für dieses spezielle Problem nur solche Wege existieren, welche von einer der Ecken D oder E starten und bei der anderen enden.

Weil jeder Weg, der bei D beginnt, bei E endet, berechnen wir die Liste aller von D ausgehenden Wege, zusammen mit ihrer Umkehrung. Hierfür können wir die Funktion `reverse` aus der Haskell-Standardfunktionsbibliothek verwenden, welche eine Liste auf ihre Umkehrung abbildet.

```
nikolaus = zb 'D'
           [('A','B'), ('A','C'), ('B','C'), ('B','D'),
            ('B','E'), ('C','D'), ('C','E'), ('D','E'),
            ('D','F'), ('D','G'), ('E','F'), ('E','G'),
            ('F','G'), ('F','H'), ('G','H')]
            [] []

ergebnis = [x | weg <- wegeInBaum nikolaus,
                x <- [weg, reverse weg]]
```

Damit ist das Problem an sich gelöst; das gesamte Haskell-Programm ergibt sich durch Aneinanderreihung der in obigem Text auftauchenden Definitionen. Wir werfen nun noch einen Blick auf die Programmausführung.

Um einen Eindruck zu bekommen, wie dieses Programm einen Zugbaum berechnet, sehen wir uns ein einfaches Beispiel an. Der einfachste doppelkanten- und schleifenfreie Graph, der auf zwei Arten gezeichnet werden kann, ist ein Dreieck. Wir ersetzen die Definition von `nikolaus` durch

```
nikolaus = zb 'A'
           [('A','B'), ('B','C'), ('C','A')]
           [] []
```

[4] "Wege in kleineren Bäumen"

Eine Besonderheit rein funktionaler Sprachen ist, daß sie *faul* (engl.: *lazy*) sind: Im Gegensatz zu C, Java oder auch näheren Verwandten wie Scheme oder SML muß für eine Definition wie diese noch nichts berechnet werden. Werte werden immer erst dann ausgerechnet, wenn ihre Kenntnis zur Berechnung des Endergebnisses unvermeidlich wird.

Nehmen wir an, wir veranlassen etwa im Haskell-System `hugs` durch Eingabe von `nikolaus` die Ausgabe des kompletten Zugbaums:

```
Main> nikolaus

Zuege ('A',[Zuege ('C',[Zuege ('B',[Ende 'A'])]),
            Zuege ('B',[Zuege ('C',[Ende 'A'])])])
```

Bis diese Ausgabe auf dem Bildschirm erschienen ist, sind eine Unmenge elementarer Rechenschritte[5] abgelaufen. Die grundsätzliche Struktur der Berechnung soll kurz skizziert werden.

Zunächst wird die Definition von `nikolaus` zu Rate gezogen. Diese besagt, daß die Funktion `zb` auszuwerten ist. Haskell übergibt anstelle der vier Funktionsargumente vier "Versprechen", bei Bedarf die Funktionsargumente auszurechnen. Wenn die auszuwertende Funktion einen Parameter überhaupt nicht verwendet, wird das zugehörige Versprechen einfach nicht eingelöst. Wäre der Wert gleich berechnet worden, so wäre nur Rechenzeit vergeudet worden. Wird ein- und dasselbe Versprechen an mehreren Stellen benötigt, so wird es beim ersten Einlösen durch den berechneten Wert ersetzt. Dadurch werden unnötige Mehrfachberechnungen vermieden. "Faule" Auswertung ist insbesondere im Zusammenhang mit unendlich großen Strukturen wichtig. Weil unsere Bäume grundsätzlich nur endlich groß sind, kann sich diese Art der Auswertung bei unserem Problem allenfalls auf die Effizienz auswirken. Deswegen wird im folgenden nicht immer streng zwischen Versprechen und Wert unterschieden.

Die vier abschnittsweisen Definitionen von `zb` werden von oben nach unten durchsucht. Hierfür muß zumindest die Struktur der Funktionsparameter bekannt sein, weswegen einige Versprechen eingelöst werden müssen. Weil die Liste der noch nicht betrachteten unbenutzten Kanten nicht leer ist, werden die für diesen Fall zutreffenden Definitionen — Zeile 1 und 2 — übergangen. Weil jene Liste auch nicht genau eine einzige Kante enthält, wird auch die Definition in Zeile 3 übergangen. Die letzte der vier abschnittsweisen Definitionen — sie beginnt in Zeile 7 — paßt zu unseren Funktionsparametern. Die Muster auf der linken Seite legen die Werte einiger Variablen auf der rechten Seite fest:

[5] genau 1905

```
se  ≡  'A'
k   ≡  ('A','B')
a   ≡  'A'
b   ≡  'B'
rk  ≡  [('B','C'),('C','A')]
bk  ≡  []
bf  ≡  []
```

Von der nachfolgenden Fallunterscheidung trifft der erste Fall (Zeile 8) zu. Das Ergebnis ist also `weiter 'B'`. Zur Berechnung dieses Werts wird für die Fallunterscheidung in Zeile 13 ein mit `kind` bezeichnetes Zwischenergebnis benötigt. Dessen Wert ist:

```
zb 'B' [('B','C'),('C','A')] [] []
```

Da auch alle folgenden Auswertungen nach den oben dargestellten Grundprinzipien ablaufen, soll die weitere Berechnung anhand einiger wesentlicher Zwischenschritte skizziert werden:

```
zb 'A'
   [('A','B'), ('B','C'), ('C','A')]
   []
   []
```

= (Zeile 14)

```
zb 'A'
   [('B','C'), ('C','A')]
   [('A','B')]
   [zb 'B'
       [('B','C'), ('C','A')]
       []
       []
   ]
```

= (Zeile 10)

```
zb 'A'
   [('C','A')]
   [('B','C'), ('A','B')]
   [zb 'B'
```

```
        [('B','C'), ('C','A')]
        []
        []
    ]
```

= (Zeile 14)

```
  zb 'A'
     []
     [('C','A'), ('B','C'), ('A','B')]
     [
      zb 'C'
         [('B','C'), ('A','B')]
         []
         [],
      zb 'B'
         [('B','C'), ('C','A')]
         []
         []
     ]
```

= (Zeile 2)

```
  Zuege ('A', [zb 'C' [('B','C'), ('A','B')] [] [],
               zb 'B' [('B','C'), ('C','A')] [] []])
```

= (Zeile 14)

```
  Zuege ('A', [zb 'C'
                  [('A','B')]
                  [('B','C')]
                  [zb 'B' [('A','B')] [] []],
               zb 'B' [('B','C'), ('C','A')] [] []])
```

= (Zeile 10)

```
  Zuege ('A', [zb 'C'
                  []
                  [('A','B'), ('B','C')]
                  [zb 'B' [('A','B')] [] []],
               zb 'B' [('B','C'), ('C','A')] [] []])
```

= (Zeile 2)

```
Zuege ('A', [Zuege ('C', [zb 'B' [('A','B')] [] []]),
             zb 'B' [('B','C'), ('C','A')] [] []])
```

= (Zeile 5)

```
Zuege ('A', [Zuege ('C', [Zuege ('B', [Ende 'A'])]),
             zb 'B' [('B','C'), ('C','A')] [] []])
```

= (...)

```
Zuege ('A', [Zuege ('C', [Zuege ('B', [Ende 'A'])]),
             Zuege ('B', [Zuege ('C', [Ende 'A'])])])
```

2.5 Das Geheimnis der 3 Karten

2.5.1 Aufgabenstellung

Stelle dir einen Kartenstapel von $2k$ ($k \in N$) Karten vor, die von oben nach unten durchnumeriert sind von 1 bis $2k$. Die oberste Karte ist also Karte 1.

Der Kartenstapel wird nun gemischt nach folgendem Verfahren:
Bei einer m-Mischung ($m \in N$ und $m \leq k$) werden die obersten m Karten abgehoben und mit den nächsten m Karten abwechselnd neu aufgelegt, beginnend mit den nicht abgehobenen Karten. Der Rest des Stapels bleibt unverändert.

Beispielsweise sieht für $k = 8$ der Kartenstapel 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 nach einer 6-Mischung so aus:
7, 1, 8, 2, 9, 3, 10, 4, 11, 5, 12, 6, 13, 14, 15, 16

Bei einer vollständigen Mischung eines Kartenstapels aus $2k$ Karten wird nacheinander eine 1-Mischung, eine 2-Mischung, eine 3-Mischung, ..., eine k-Mischung in dieser Reihenfolge ausgeführt.

In obigem Beispiel ergibt sich (1. Spalte = Anzahl Mischungen):

	1,	**2**,	**3**,	**4**,	**5**,	**6**,	**7**,	**8**,	**9**,	**10**,	**11**,	**12**,	**13**,	**14**,	**15**,	**16**
1	2,	1,	**3**,	**4**,	**5**,	**6**,	**7**,	**8**,	**9**,	**10**,	**11**,	**12**,	**13**,	**14**,	**15**,	**16**
2	3,	2,	4,	1,	**5**,	**6**,	**7**,	**8**,	**9**,	**10**,	**11**,	**12**,	**13**,	**14**,	**15**,	**16**
3	1,	3,	5,	2,	6,	4,	**7**,	**8**,	**9**,	**10**,	**11**,	**12**,	**13**,	**14**,	**15**,	**16**
4	6,	1,	4,	3,	7,	5,	8,	2,	**9**,	**10**,	**11**,	**12**,	**13**,	**14**,	**15**,	**16**
5	5,	6,	8,	1,	2,	4,	9,	3,	10,	7,	**11**,	**12**,	**13**,	**14**,	**15**,	**16**
6	9,	5,	3,	6,	10,	8,	7,	1,	11,	2,	12,	4,	**13**,	**14**,	**15**,	**16**
7	1,	9,	11,	5,	2,	3,	12,	6,	4,	10,	13,	8,	14,	7,	**15**,	**16**
8	4,	1,	10,	9,	13,	11,	8,	5,	14,	2,	7,	3,	15,	12,	16,	6

Aufgabe:
Schreibe ein Programm, das nach Eingabe von k ($k \geq 2$) einen Kartenstapel mit $2k$ Karten vollständig mischt. Während der Mischung sollen einige Informationen protokolliert werden, so daß das Programm die folgenden Fragen beantwortet:

1. Welches sind die drei obersten Karten nach der vollständigen Mischung?
2. Welches sind die drei Karten, die während des Mischens am häufigsten oben lagen? (Bei gleicher Häufigkeit interessiert die niedrigere bzw. niedrigste Kartennummer.)
3. Bei der wievielten Mischung lag die oberste Karte zum erstenmal oben?
4. Wie oft lag die oberste Karte bei den Mischungen insgesamt oben?

Gib uns fünf Lösungen an für verschiedene, möglichst große k, darunter $k = 500$.

2.5.2 Lösungsidee

Wir benötigen zwei Arrays, *Karten* und *Statistik*. *Karten* enthält den aktuellen Kartenstapel, d. h. vor dem ersten Mischen gilt $karten[i] = i$. Im Verlauf des Mischens ändert sich das natürlich.

Im Array *Statistik* bleiben die Karten dagegen unverändert in der Reihenfolge $1, \ldots, 2k$ stehen; zwei Record-Komponenten *wieoft* und *wann* geben für jede Karte an, wie oft sie die oberste Karte des Stapels war und wann sie zum erstenmal oben lag. Die Werte *wann* und *wieoft* werden für alle Karten mit 0 initialisiert, denn der Ausgangsstapel hat keinen Einfluß auf die Statistik. Insbesondere hat die 1 nicht bereits einmal oben gelegen, bevor das erste Mischen erfolgt.

Für das Mischen verwenden wir ein Hilfsarray *NeueKarten*, in dem der gemischte Stapel zwischengespeichert wird.

Eine m-Mischung geht dann folgendermaßen vor sich: Mit zwei Hilfszeigern *abanfang* und *abmitte* wird das Array *Karten* durchlaufen. Der Ausdruck "Zeiger" ist dabei nicht ganz korrekt, es handelt sich nur um zwei Integer-Variable, die Indizes im Array angeben. Die Variable *abanfang* wird mit 1 initialisiert, *abmitte* mit $m + 1$. Nun werden in das Array *NeueKarten* abwechselnd die Karten eingetragen, die *abmitte* und *abanfang* angeben. Zum Schluß wird das Ergebnis des Mischens in das Array *Karten* geschrieben, um dort wieder den aktuellen Kartenstapel vorliegen zu haben. Schließlich wird die Statistik aktualisiert, indem überprüft wird, welche Karte oben liegt. Deren Zähler *wieoft* wird erhöht, und wenn die Karte zum erstenmal oben liegt, wird die Nummer der Mischung, also m, in *wann* eingetragen.

Falls m kleiner als k ist, bleibt ein Teil des ursprünglichen Stapels unverändert. Das wird korrekt behandelt, wenn man vor Beginn des Mischens *NeueKarten* mit *Karten* initialisiert oder indem nach dem Mischen nur die ersten $2m$ Elemente aus *NeueKarten* in *Karten* übertragen werden. In *Karten* steht ja noch der unverändert gebliebene Teil des Kartenstapels. Die zweite Möglichkeit ist effizienter, denn es erfolgen keine überflüssigen Zuordnungen.

Nach dem vollständigen Mischen muß nun die Statistik ausgegeben werden. Die drei obersten Karten liefert das Array *Karten*; wie oft die oberste Karte oben lag und wann zum erstenmal liefern die Einträge *Statistik*[*Karten*[*1*]].*wieoft* und *Statistik*[*Karten*[*1*]].*wann*. Das Array *Statistik* wird nun dreimal nach dem Maximum der *wieoft*-Werte durchsucht, und die entsprechenden Karten werden ausgegeben.

Lösung für $k = 500$:
Die drei obersten Karten nach der vollständigen Mischung sind 214, 90 und 883.
Die 214 lag insgesamt zweimal oben, zum erstenmal bei der 439. Mischung.
Am häufigsten oben lagen die 14 (sechsmal), die 1 (fünfmal) und die 16 (ebenfalls fünfmal).

Weitere Lösungen:

$k = 1000$:
Die drei obersten Karten nach der vollständigen Mischung sind 114, 725 und 547.
Die 114 lag insgesamt zweimal oben, zum erstenmal bei der 271. Mischung.
Am häufigsten oben lagen die 1, die 14 und die 6.

$k = 1500$:
Die drei obersten Karten nach der vollständigen Mischung sind 258, 410 und 2430.
Die 258 lag insgesamt zweimal oben, zum erstenmal bei der 537. Mischung.
Am häufigsten oben lagen die 1, die 14 und die 29.

$k = 2000$:
Die drei obersten Karten nach der vollständigen Mischung sind 1898, 3320 und 3191.
Die 1898 lag insgesamt zweimal oben, zum erstenmal bei der 1593. Mischung.
Am häufigsten oben lagen die 38, die 1 und die 14.

2.5.3 Programmtext

```
program Kartenstapel;

const max = 5000;
type KartenType = array[1..max] of integer;
var Karten : KartenType;
    k : integer;
    Statistik : array[1..max] of record
                                    wieoft, wann: integer
                                end;

procedure init;
var i: integer;
begin
  write('Vollständig␣mischen␣für␣welches␣k?␣');
  readln(k);
  for i := 1 to 2*k do
```

```
  begin
    Karten[i] := i;
    Statistik[i].wieoft := 0;
    Statistik[i].wann := 0
  end
end;

procedure mischen(m:integer);
var NeueKarten : Kartentype;
    i, abanfang, abmitte : integer;
begin
  abanfang := 1;
  abmitte := m+1;
  i := 1;
  repeat
    NeueKarten[i] := Karten[abmitte];
    inc(abmitte);
    inc(i);
    NeueKarten[i] := Karten [abanfang];
    inc(abanfang);
    inc (i);
  until abanfang > m;
  for i := 1 to 2*m do
    Karten[i] := NeueKarten[i]
end;

procedure StatistikAuffrischen(aktuell:integer);
begin
  inc(Statistik[Karten[1]].wieoft);
  if Statistik[Karten[1]].wieoft = 1 then
    Statistik[Karten[1]].wann := aktuell
end;

procedure VollstaendigMischen;
var i : integer;
begin
  for i := 1 to k do
  begin
    mischen(i);
    StatistikAuffrischen(i)
  end
end;
```

```
procedure StatistikAusgeben;
var haeufig : record
                wer, wieoft : integer
              end;
    i,j,h : integer;
begin
  writeln('Endgültiger Stand: ');
  writeln('Oberste drei Karten: ');
  for i := 1 to 3 do
    write(Karten[i]:4,' ');
  writeln;
  writeln('Die oberste Karte ',Karten[1],' lag bei der ',
    Statistik[Karten[1]].wann,'. Mischung zum ersten Mal oben ');
  writeln('und lag insgesamt ',Statistik[Karten[1]].wieoft,' mal oben.');
  write('Die am häufigsten obenliegenden Karten waren');
  for i := 1 to 3 do
  begin
    writeln;
    haeufig.wer := 0;
    haeufig.wieoft := 0;
    for j := 1 to 2*k do
      if Statistik[j].wieoft > haeufig.wieoft then
      begin
        haeufig.wer := j;
        haeufig.wieoft := Statistik[j].wieoft
      end;
    write('Karte ',haeufig.wer,' mit ',haeufig.wieoft,' Vorkommen');
    if i = 3 then write('.')
             else write(',');
    Statistik[haeufig.wer].wieoft := 0
  end;  {for i := 1 to 3}
  writeln
end;

begin
  init;
  VollstaendigMischen;
  StatistikAusgeben;
  readln
end.
```

3 12. Wettbewerb: Die Aufgaben der 2. Runde

3.1 Mustererkennung

3.1.1 Aufgabenstellung

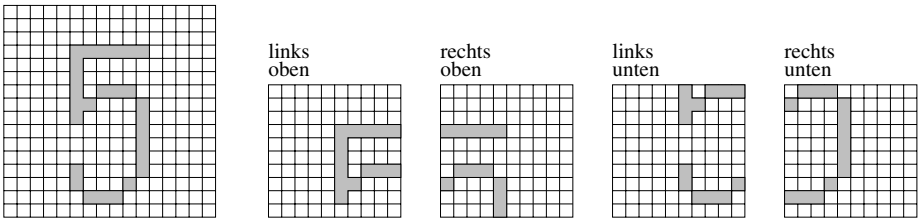
Zum Ausfüllen von Formularen werden Ziffern in Kästchen mit einem 16×16-Punktraster handschriftlich eingetragen unter Einhaltung der folgenden Bedingungen:

Höhe: 10 bis 16 Punkte

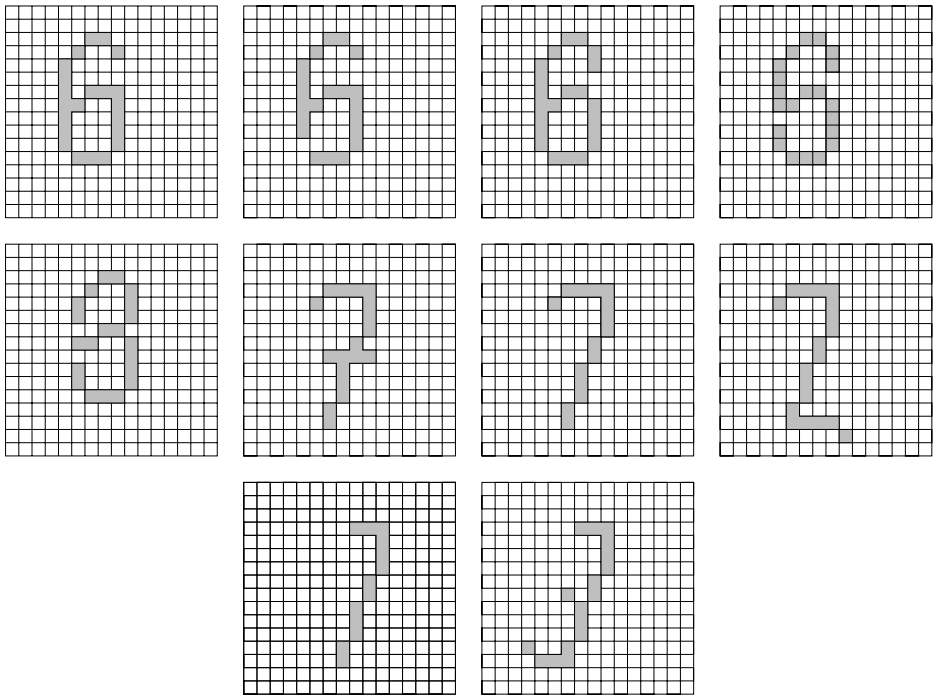
Breite: 1/3 bis 2/3 der Höhe

Dicke der Schrift: 1 bis 3 Punkte

Das 16×16-Punktraster wird in vier, sich überlappende Quadrate von 10×10 Punkten aufgeteilt, die einzeln ausgewertet werden. Die Auswertung der vier Quadrate liefert jeweils eine Liste von charakteristischen Merkmalen wie etwa: gerade Linien, geschlossene Figur usw.



1. Entwerfen Sie ein kleines Sortiment von charakteristischen Merkmalen, das zur Klassifizierung der 10 Ziffern ausreicht.
2. Geben Sie für jede Ziffer und jedes ihrer vier Quadrate die Merkmale an, die vorliegen müssen, damit die Ziffer klassifiziert werden kann.
3. Dokumentieren Sie die Grenzen des Verfahrens.
4. Schreiben Sie ein Programm, das das 16×16 Punktraster in die vier Quadrate aufteilt, die erkannten Merkmale und die zugehörige Ziffer bzw. eine Fehlermeldung ausgibt.
5. Dokumentieren Sie an den folgenden Beispielen, wie Ihr Programm zu einer Entscheidung kommt. Es muß nicht in allen Fällen zu einer zweifelsfreien Entscheidung kommen. Senden Sie uns außerdem noch fünf weitere typische Beispiele.



3.1.2 Erläuterungen zur Aufgabe

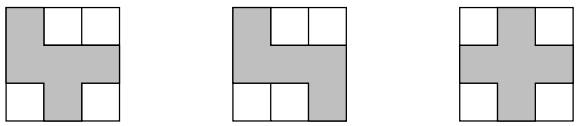
Die zentrale Schwierigkeit der Aufgabe besteht in der Erstellung eines geeigneten Satzes von Merkmalen, anhand dessen die zehn Ziffern erkannt werden können. Dieser Merkmalssatz muß zum Teil gegensätzliche Forderungen erfüllen. Einerseits müssen die Merkmale allgemein genug sein, um verschiedene Schreibweisen tolerieren zu können, andererseits dürfen aber völlig unleserliche Zeichen nicht fälschlich identifiziert werden. Ein wichtiger Punkt ist, daß die Merkmalserkennung durch ein Programm möglichst effizient durchgeführt werden soll. Es müssen also Merkmale definiert werden, deren Bestimmung an einem konkreten Muster nicht zu rechenaufwendig ist.

Die Aufgabenstellung sieht vor, ein 16×16-Punktraster nicht als Ganzes, sondern zerlegt in vier sich überlappende 10×10-Punktraster zu betrachten. Bei der Charakterisierung der Merkmale kann man also von Ziffern als Gesamtbild absehen und sich auf Teilaspekte konzentrieren. In den vier Teilrastern können durchaus verschiedene Ziffern wahrscheinlich erscheinen; erst eine Kombination der Teilergebnisse sollte zu einem möglichst eindeutigen Ergebnis kommen. Dafür kann es notwendig sein, die Teilergebnisse unterschiedlich zu gewichten.

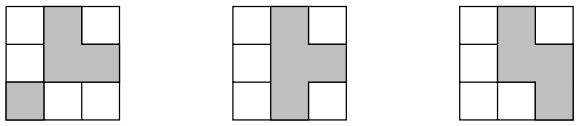
3.1.3 Entwurf charakteristischer Merkmale[5]

Als Merkmale zur Erkennung bieten sich einfache geometrische Muster wie Linien, Linienenden, (teilweise) geschlossene Figuren, Schlaufen, Verzweigungen usw. an. Dabei kann jedem Merkmal eine Position und eine Orientierung zugeordnet werden. Zur Erkennung eines Merkmals in einem konkreten Muster wird je nach Merkmal ein geeignetes Verfahren eingesetzt. *Linienenden*, *Verzweigungen* und *Kreuzungen* lassen sich durch einfache Bitmustervergleiche leicht bestimmen. *Linien* bestehen aus einem Start- und einem Endpunkt, die durch eine Reihe benachbarter Punkte (auch diagonal) miteinander verbunden sind. Weil bei der Mustererkennung grundsätzlich mit „verrauschter“ Information zu rechnen ist, sollten auch nicht ganz zusammenhängende Linien als eine Linie erkannt werden. Dazu kann ein Grenzwert festgelegt werden, der angibt, wieviel Prozent der Punkte einer vollständigen Linie vorhanden sein müssen.

An einer *Kreuzung* schneiden sich zwei Linien. Zur Erkennung genügt die Betrachtung eines 3×3-Punktrasters: Eine Kreuzung liegt genau dann vor, wenn eine horizontale oder vertikale Linie durch zwei Punkte an gegenüberliegenden Seiten ergänzt wird. Zur Verdeutlichung dienen die drei unten stehenden Beispiele. Alle anderen Kreuzungen lassen sich durch Drehungen und Spiegelungen aus diesen drei Beispielen herstellen.



An einer *Verzweigung* gehen vom Ende einer Linie zwei andere Linien aus. Genau wie bei einer Kreuzung kann auch eine Verzweigung ganz lokal durch Betrachtung von 3×3-Punktrastern erkannt werden. Auch hier kann man drei Beispiele angeben, aus denen sich durch Drehung und Spiegelung alle Möglichkeiten für Verzweigungen erzeugen lassen.



Ob ein Punkt in einer *teilweise* oder *vollständig geschlossenen Figur* liegt, kann man feststellen, indem man zählt, in wie vielen der vier Grundrichtungen (links, rechts, oben, unten) sich in beliebigem Abstand “Nachbarpunkte” befinden. Wenn ein Punkt zu einer (teilweise) geschlossenen Figur gehört, wird überprüft, welche Punkte der Nachbarschaft ebenfalls zu dieser Figur gehören. Damit kann der

[5] Die folgende Lösung beruht auf der Einsendung von Simon Trebst.

Punkt nun charakterisiert werden, ob er zu einer geschlossenen oder einer nur teilweise geschlossenen Figur gehört, in welche Richtung diese Figur geöffnet ist und ob es sich um eine kleine Figur, als *Schlaufe* (falls geschlossen) oder als *Kerbe* (falls geöffnet) bezeichnet, oder um eine größere Figur, eine *Bucht*, handelt.

Die beschriebenen Merkmale sind alle effizient in einem gegebenen Muster auffindbar. Ein einfaches Zuordnen der Merkmale zu den Ziffern ist aber problematisch, weil verschiedene Merkmale in verschiedenen Schreibweisen gänzlich fehlen oder von großer Signifikanz seinen können. Hilfreich ist die Einteilung in *notwendige* Merkmale, die in jeder Schreibweise auftauchen müssen, *zusätzliche* Merkmale, die in einigen Schreibweisen auftauchen und deutliche Hinweise geben können, *erlaubte* Merkmale, die für die Klassifikation neutral sind und sich etwa durch unpräzise Handschriften ergeben können, sowie schließlich *verbotene* Merkmale, die darauf hinweisen, daß eine bestimmte Ziffer gerade nicht vorliegt.

Unter Berücksichtigung einer Anzahl von Schreibweisen, die erkannt werden sollen, kann dann für jede Ziffer und jedes Teilraster eine Liste von Merkmalen der verschiedenen Klassen aufgestellt werden. Es genügt dabei, notwendige, erlaubte und zusätzliche Merkmale festzulegen, alle anderen Merkmale, die erkannt werden, gelten dann als verboten.

3.1.4 Ein Programm zur Klassifikation

Bevor die eigentliche Musterklassifizierung stattfindet, kann man eine Vorverarbeitung durchführen, die der Routine zur Merkmalsextraktion Zeichen in standardisierter Form vorlegt. Leicht zu realisieren sind eine Zentrierung des Zeichens in dem 16×16-Punktraster und das Entfernen einzelner, isolierter Punkte. Weniger einfach, aber für einen Bitmustervergleich notwendig, ist eine Skelettierung der Muster; dabei wird die Dicke aller Linien und Bögen auf 1 reduziert, die Eingabe wird auf ihre geometrische Gestalt zurückgeführt.

Zusammen mit einer solchen Vorverarbeitung kann ein Programm zur Mustererkennung mit folgender Grobstruktur realisiert werden:

```
Initialisiere Datenstrukturen;
Wiederhole:
  lies Zeichenraster Z;
  führe Vorverarbeitung für Z durch;
  führe Merkmalsextraktion für Z in jedem Teilraster durch;
  berechne Klassifikationen zu den erkannten Merkmalen, das sind
    gemäß gegebener Merkmalslisten passende Ziffern;
  kombiniere die vier Klassifikationen zur Gesamtklassifikation;
  gib als Ergebnis die erkannte Ziffer oder NICHT ERKANNT aus
bis alle Zeichen abgearbeitet wurden.
```

Jede Teilklassifizierung sollte mit einem numerischen Wert versehen werden, der angibt, mit welcher Sicherheit das angegebene Zeichen erkannt wurde. Es ist durchaus auch sinnvoll, solche Zuversichtswerte für alle zehn Ziffern anzugeben und für die Gesamtklassifizierung zu addieren. Hat kein Zeichen eine hinreichend große Sicherheit oder sind mehrere Zeichen ähnlich sicher erkannt worden, so sollten entweder alle erkannten Zeichen oder "Nicht erkannt" ausgegeben werden.

Für die Güte des Verfahrens ist entscheidend, die Merkmalssätze der Ziffern so zu definieren, daß sich nicht zu viele Überschneidungen mit anderen Ziffern ergeben. Beschränkt man sich auf wenige charakteristische Merkmale, so sind diese in der Regel bei vielen Ziffern gleich und es ist keine eindeutige Entscheidung möglich.

3.1.5 Beispiel

Im folgenden beschreiben wir einen möglichen charakteristischen Merkmalssatz für die 6. Dabei bedeuten Aufzählungen in einer Zeile, daß jeweils nur eines dieser Merkmale auftreten muß. Notwendige Merkmale werden durch ein ! am Zeilenanfang beschrieben, zusätzliche Merkmale durch ein + und erlaubte Merkmale durch ein ?. Die Abkürzungen für die Himmelsrichtungen geben an, in welche Richtung die entsprechende Figur orientiert bzw. geöffnet ist, und schließlich wird noch die allgemeine Positionierung der Figur im betrachteten Quadrat angegeben.

6, linkes oberes Quadrat:

```
! Bucht O, Kerbe O, Linienende N oben;
! Linie N links;
! Verzweigung O, Kreuzung;
! Bucht S unten, Kerbe S unten, Bucht M unten; (M: zur Mitte)
+ Linienende NO oben, Linienende O oben, Linienende SO oben;
+ Kerbe N, Kerbe NO;
? Linie N rechts;
? Linie O oben;
? Linie O unten;
? Linie NO;
```

6, rechtes oberes Quadrat:

```
! Bucht OW, Bucht O;
! Linienende NO oben, Linienende O oben, Linienende SO oben;
! Bucht M, Bucht S, Kerbe S;
+ Verzweigung O, Kreuzung;
? Linie N;
? Linie O oben;
? Linie O unten;
```

6, linkes unteres Quadrat:

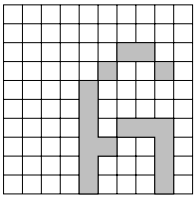
```
! Bucht G, Bucht O, Bucht M; (G: geschlossen)
! Linie N;
! Verzweigung O, Kreuzung;
+ Kerbe N oben;
? Linie O oben;
? Linie O unten;
? Linie N rechts;
? Linie SO;
```

6, rechtes unteres Quadrat:

```
! Bucht G, Bucht W, Bucht M;
? Linie O oben;
? Linie O unten;
? Linie N links;
? Linie N rechts;
? Linie NO;
```

Bei der 1. Ziffer des Beispiels werden links oben folgende Merkmale erkannt:

links oben:

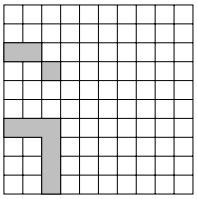


Erkannte Merkmale:

```
Linie N links        Bucht O oben
Linie N rechts       Bucht S unten
Linie O unten
Linienende SO oben
Verzweigung O
```

Einige Merkmale könnten auch zur Ziffer 4 oder 5 gehören, aber es treten auch Merkmale auf, die für beide Ziffern nicht erlaubt sind. Damit lautet das Zwischenergebnis für das linke obere Quadrat 6.

rechts oben:

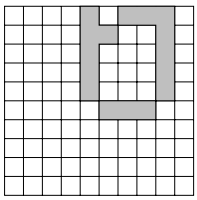


Erkannte Merkmale:

```
Linie N links
Linienende SO oben
Bucht OW oben
Bucht M unten
```

Es werden alle notwendigen Merkmale für die Ziffern 0, 1, 5 und 6 erkannt. Für 0 und 1 treten aber auch nicht erlaubte Merkmale auf, so daß sich als Zwischenergebnis für das rechte obere Quadrat ergibt: 6 5

links unten:

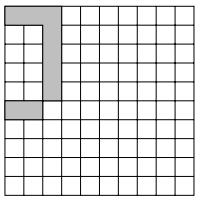


Erkannte Merkmale:

```
Linie N links        Bucht G oben
Linie N rechts
Linie O oben
Verzweigung O
Kerbe N oben
```

Hier wurden nicht nur die für die 6 notwendigen Merkmale entdeckt, sondern auch alle für die 8 und einige der für die 7 notwendigen Merkmale. Für die 7 treten jedoch einige nicht erlaubte Merkmale auf; außerdem gibt es mehr zusätzliche Merkmale, die auf die 8, nicht aber auf die 7 zutreffen, die 7 wird also nicht als mögliche Ziffer akzeptiert. Das Zwischenergebnis für das linke untere Quadrat lautet: 8 6

rechts unten:



Erkannte Merkmale:

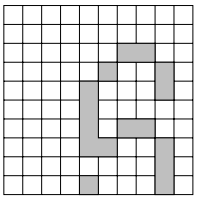
```
Linie N links
Bucht W oben
```

Hier lautet das Zwischenergebnis: 5 3 8 6

Als Gesamtergebnis aus den vier Teilen erhält man: 6

Als weiteres Beispiel betrachten wir noch das vierte Muster aus der Aufgabenstellung. Im linken oberen Quadrat werden die folgenden Merkmale erkannt:

links oben:

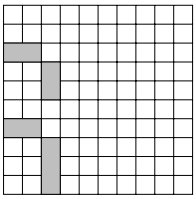


Erkannte Merkmale:

```
Linie N links        Kerbe W unten
Linie O unten        Kerbe S unten
Linienende S oben
Bucht O oben
Kerbe S unten
```

Die drei Merkmale `Kerbe S unten, Kerbe W unten, Kerbe S unten` führen hier zu einer negativen Bewertung der Ziffern 5 und 6, aber zu einer positiven Bewertung für 8, so daß das Zwischenergebnis für das linke obere Quadrat 8 lautet.

rechts oben:

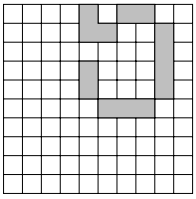


Erkannte Merkmale:

```
Linienende S oben
Bucht OW oben
Bucht M unten
```

Es werden alle notwendigen Merkmale für die Ziffern 0, 5 und 6 erkannt. Für 0 treten aber wiederum nicht erlaubte Merkmale auf, so daß sich als Zwischenergebnis für das rechte obere Quadrat ergibt: 6 5

links unten:

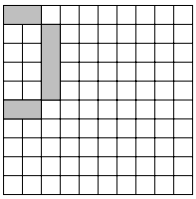


Erkannte Merkmale:

```
Linie O oben
Linie N rechts
Linienende N unten
Kerbe N oben
Bucht W oben
```

Das Zwischenergebnis für das linke untere Quadrat lautet: 3 5

rechts unten:



Erkannte Merkmale:

```
Linie N links
Bucht W oben
```

Hier lautet das Zwischenergebnis wie schon beim 1. Beispiel: 5 3 8 6

Das Gesamtergebnis lautet: 5 6, d. h. das Programm kommt bei diesem Muster zu keinem eindeutigen Ergebnis.

3.1.6 Erweiterungen

Mit den vorgestellten Merkmalen lassen sich natürlich auch Buchstaben charakterisieren. Stellt man dem Programm entsprechende Merkmalssätze zur Verfügung, so kann es ohne weitere Modifikation auch zur Erkennung von Buchstaben eingesetzt werden.

3.2 Hamiltons Roboter

3.2.1 Aufgabenstellung

Gegeben sind n Punkte $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$ in der Ebene, wobei alle Koordinaten ganzzahlig sind. Ein Roboter soll beginnend bei P_1 alle Punkte $P_2, \ldots, P_n$ in beliebiger Reihenfolge genau einmal aufsuchen und am Ende wieder bei P_1 anlangen. Die Bewegungsfähigkeit des Roboters ist allerdings eingeschränkt und durch die folgenden Anweisungen der Roboterprogrammiersprache definiert:

`Richtung` $\mathtt{x}_i$ $\mathtt{y}_i$ Der Roboter dreht sich in Richtung auf den Punkt (x_i, y_i), wobei (x_i, y_i) aus der eingegebenen Liste von Punkten ist. Diese Anweisung darf nur ein einziges Mal am Anfang eines Roboterprogramms verwendet werden.

`Gehe zu` $\mathtt{x}_i$ $\mathtt{y}_i$ Wenn der Roboter von seiner augenblicklichen Position und in der augenblicklichen Richtung in gerader Linie den Punkt (x_i, y_i) aus der eingegebenen Liste erreichen kann, geht er dorthin und führt danach den nächsten Befehl des Programms aus. Anderenfalls bleibt er stehen und verweigert den weiteren Dienst.

`Links` Der Roboter dreht sich um 90° nach links.

`Rechts` Der Roboter dreht sich um 90° nach rechts.

`Stop` Der Roboter schaltet sich aus.

Aufgabe: Schreibe ein Programm, welches folgendes leistet:

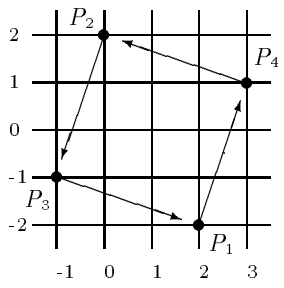
1. Einlesen der Zahl n der Punkte und der n Koordinatenpaare.
2. Ermitteln, ob es eine Rundreise des Roboters unter den gegebenen Bedingungen gibt oder nicht.
3. Ausgabe eines Roboterprogramms für die Rundreise. Wenn es keine Rundreise gibt, besteht das Programm nur aus der Anweisung `Stop`.

Beispiel: Für die Eingabe

```
4
2 -2
0 2
-1 -1
3 1
```

gibt es eine Rundreise; das entsprechende Roboterprogramm und die Rundreise sehen so aus:

```
Richtung 3 1
Gehe zu 3 1
Links
Gehe zu 0 2
Links
Gehe zu -1 -1
Links
Gehe zu 2 -2
Stop
```



3.2.2 Erläuterungen zur Aufgabe

Die Aufgabe sieht vor, n Punkte bei festem Startpunkt in beliebiger Reihenfolge genau einmal zu besuchen und zum Startpunkt zurückzukehren. Man kann die n Punkte mit den Verbindungen zwischen ihnen als Graph modellieren. Dieser Graph ist fast vollständig, da der Roboter von jedem Knoten zu jedem beliebigen anderen Knoten gehen kann, es sei denn, er überläuft dabei einen Knoten. (Da ein überlaufener Knoten irgendwo in der Rundreise explizit auftreten muß, wäre er zweimal besucht worden.) Bei kollinearen Knoten gibt es daher immer nur die Kanten zwischen direkt benachbarten Knoten. Die Aufgabe besteht nun darin, in diesem Graph einen Hamilton-Kreis zu finden, wenn es einen gibt. Im allgemeinen ist dieses Problem schwierig, so ist etwa das Problem zu entscheiden, ob es in einem Graphen einen Hamilton-Kreis gibt, NP-vollständig und darum vermutlich nicht mit polynomieller Rechenzeit lösbar. Im konkreten Fall kann man im zu betrachtenden Graph viele Kanten entfernen und das Problem so etwas vereinfachen. Nicht zu große Eingaben bleiben damit trotz der Schwere des Problems handhabbar. Entfernt man nach der Wahl des Startknotens und der ersten Kante alle Kanten, die der Roboter aufgrund seiner eingeschränkten Bewegungsfähigkeit nicht mehr benutzen kann, dann kann man den so erhaltenen Graphen als Gittergraphen auffassen; der Graph wird in ein quadratisches Raster eingebettet, alle Knoten liegen auf Rasterkreuzungen, alle Kanten verlaufen auf Rasterkanten. Das Problem, für einen derart eingeschränkten Graphen einen Hamilton-Kreis zu finden, bleibt allerdings NP-vollständig.

Der Graph in Abbildung 3.1 ist ein Beispiel für eine mit nur 13 Knoten sehr kleine Eingabe, der man schon fast direkt ansehen kann, daß sie einen Hamilton-Kreis enthält, der den besonderen Anforderungen der Aufgabenstellung genügt. Wir werden noch Kriterien formulieren, um diese Entscheidung möglichst schnell treffen zu können.

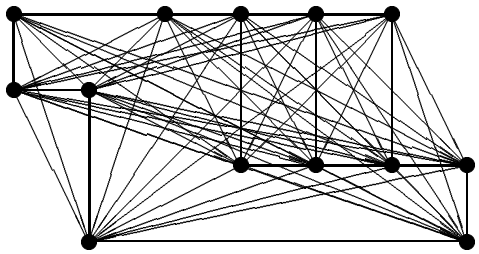


Abbildung 3.1: Ein fast vollständiger Graph mit 13 Knoten

3.2.3 Lösungsideen

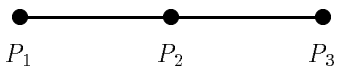
Eine naheliegender Lösungsansatz benutzt einen Backtracking-Algorithmus, um systematisch alle möglichen Wege auszuprobieren, bis entweder ein Hamilton-Kreis gefunden wurde oder alle Wege erfolglos probiert wurden und man sicher sein kann, daß es keine solche Rundreise gibt. Weil es bei n Punkten $n!$ verschiedene Rundreisen geben kann, lohnt es sich, etwas Aufwand zu betreiben, um die Anzahl der Rundreisen, die betrachtet werden müssen, zu reduzieren. Dabei kann man zwischen Möglichkeiten zum Ausschluß von Rundreisen unterscheiden, die allgemein in allen Graphen durchgeführt werden können und solchen, die sich durch die eingeschränkte Bewegungsfähigkeit des Roboters ergeben.

Bei einem Hamilton-Kreis muß jeder Knoten genau einmal betreten und auf anderem Weg wieder verlassen werden. Der Roboter kann sich nur in 90° Schritten drehen, darum kann ein Punkt nur über eine Kante verlassen werden, die zu der Kante, über die er erreicht wurde, parallel oder senkrecht steht.[6] Man kann darum für jeden Punkte alle Paare von Kanten prüfen, die mit ihm verbunden sind. Kanten, für die nicht wenigstens eine andere Kante parallel oder senkrecht ist, können aus dem Graph entfernt werden, sie kommen für Rundreisen nicht in Frage. Wir führen diese Operation zunächst jedoch nicht durch, sondern dünnen den Graphen später stärker aus, wenn der 2. Punkt der Rundreise bereits gewählt wurde.

Parallele Kanten verdienen noch eine besondere Betrachtung. Wenn zwei Kanten einen Punkt in dieselbe Richtung verlassen, so muß die längere Kante aus dem

[6] Mit *parallel* ist hier ausdrücklich nicht die graphentheoretische Bedeutung gemeint, sondern parallel bezieht sich hier auf die Orientierung der Kanten in der Ebene.

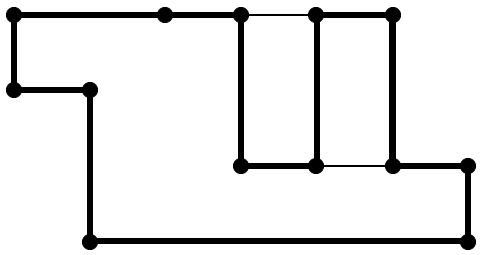
Graph entfernt werden; sie darf auf einer Rundreise nicht benutzt werden, wie man in der folgenden Abbildung leicht sieht.



Bliebe die Kante (P_1, P_3) im Graphen, könnte eine Rundreise P_1, P_3, P_2, P_1 errechnet werden, die jedoch keine gültige Rundreise ist.

Nach der Wahl des zweiten Knotens der Rundreise können noch weitere Kanten aus dem Graph entfernt werden. Alle Kanten, die weder parallel noch senkrecht zur ersten Kante der Rundreise sind, können wegen der eingeschränkten Bewegungsfähigkeit des Roboters nicht benutzt werden. Man kann also nach der Wahl des ersten Kante im Backtracking-Algorithmus alle solchen Kanten entfernen, um die Suchzeit zu reduzieren. Nach Wahl der 1. Kante kann das gesamte Ausdünnen des Graphen einschließlich Entfernen nicht benutzbarer paralleler Kanten in Zeit $O(n^3)$ erfolgen.

Im Beispielgraph der Abbildung 3.1 sei der Punkt oben links der Startpunkt, also P_1. Wählt man die Kante zu seinem Nachbarn rechts als erste Kante, so ergibt sich folgender um unnötige Kanten verkleinerte Graph, die dicker gezeichneten Kanten bilden einen geeigneten Hamilton-Kreis.



Es gibt zwei Möglichkeiten, unbrauchbare parallele Kanten zu entfernen. Entweder man prüft vor der Wahl der 1. Kante (also des 2. Punktes auf der Rundreise) alle Tripel von Punkten auf Kollinearität, oder man prüft erst nach Wahl der 1. Kante, kann sich dann aber auf solche Tripel beschränken, die unter den nun eingeschränkten Bewegungsmöglichkeiten kollinear sind. Die 2. Möglichkeit geben wir in der halbformalen Programmbeschreibung an.

Unabhängig von den Besonderheiten des Roboters gibt es allgemeine Merkmale, die zeigen, daß es in einem Graph keinen Hamilton-Kreis geben kann, und die sich effizient berechnen lassen. Weil jeder Punkt genau einmal betreten und auf anderem Weg verlassen werden muß, kann es offenbar keinen Kreis geben, wenn es einen Punkt gibt, der von weniger als zwei Kanten berührt wird. Auch müssen stets alle Punkte vom Startpunkt aus erreichbar sein; wenn der Graph nicht zusammenhängend ist, kann es keinen Hamilton-Kreis geben. In Zeit $O(n^2)$ läßt sich das mit einer Tiefensuche leicht überprüfen.

Abgesehen von Möglichkeiten, die Suche nach einem Hamilton-Kreis vorzeitig abzubrechen, kann man in diesem speziellen Fall die Berechnungen noch etwas effizienter gestalten. Man kann das Koordinatensystem so transformieren, daß der Startknoten im Ursprung des Koordinatensystems liegt und die erste benutzte Kante in Richtung einer Koordinatenachse liegt. Dann kommen nur noch Schritte in Richtung einer Koordinatenachse in Frage.

3.2.4 Konkretisierung der Lösungsidee

Die oben aufgeführten verschiedenen Ideen zur Reduzierung des Zeitaufwandes können in einer halbformalen Programmbeschreibung konkretisiert werden; der Weg zu einem lauffähigen Programm ist dann nicht mehr weit.

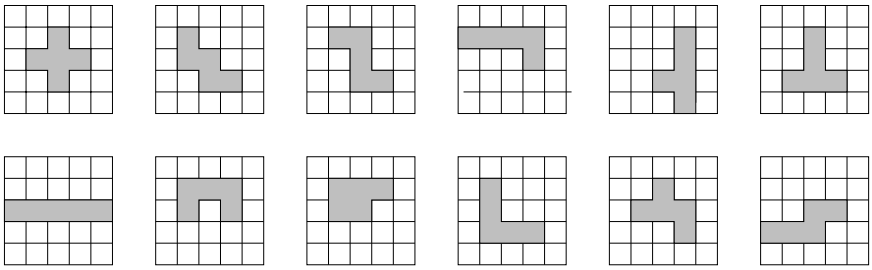
Der Graph wird als *Adjazenzmatrix* dargestellt, d. h. als eine Matrix $A[1..n, 1..n]$, in der $A[i, j]$ den Wert 1 hat, wenn es zwischen den Knoten P_i und P_j eine Kante gibt, und 0, wenn es keine solche Kante gibt. In einem Array markieren wir schon besuchte Punkte. Wir beschreiben ein Backtracking-Verfahren; die Zurücknahme der aktuellen Wahl und das Fortfahren mit der nächsten möglichen Wahl des nächsten Punktes der Rundreise beschreiben wir einfach mit der Bemerkung $\rightarrow$ `Backtracking.` Wenn ein solcher Backtrack-Schritt einmal nicht möglich ist, kann die Suche erfolglos abgebrochen werden.

```
1. Initialisiere:
1.1  lies die Anzahl der Punkte n und ihre Koordinaten ein;
1.2  initialisiere die Matrix A[1..n,1..n] mit 0 in der
       Hauptdiagonalen und mit 1 sonst;
2. Wähle 2. Punkt der Rundreise, dies sei P_i:
   markiere P_i als besucht;
2.1  dünne Graph aus (d.h. setze entsprechende Matrixeinträge auf 0):
2.1.1  entferne Kanten, die nicht parallel oder
         senkrecht zu (P_1, P_i) sind;
2.1.2  entferne Kanten, die nicht benutzbar sind,
         weil weiterer Punkt auf ihnen liegt;
3. Prüfe Graph auf Verwendbarkeit:
3.1  wenn Graph nicht zusammenhängend
     oder wenn es Punkte mit < 2 Kanten gibt:
       gehe zu 2.
4. Suche Hamiltonpfad in Graph:
4.1  wiederhole:
       wähle vom aktuellen Punkt P_i einen gemäß Matrix A
         erreichbaren und noch nicht besuchten Punkt P_j
         als nächsten aktuellen Punkt;
       markiere P_j als besucht;
       wenn kein Punkt mehr erreichbar:
       → Backtracking, d.h. wähle vom aktuellen
         Punkt aus anderen nächsten Punkt als aktuellen Punkt
     bis alle Punkte besucht oder kein neuer Punkt mehr erreichbar;
5. Beende:
     wenn alle Punkte besucht:
       wenn P_1 vom aktuellen Punkt aus erreichbar:
         gib Lösung aus;
         stop
       sonst:
         wenn noch nicht alle Punkte als 2. Punkt der Rundreise probiert:
           gehe zu 2.
         sonst: keine Lösung.
```

3.3 Pentapuzzle

3.3.1 Aufgabenstellung

Aus fünf Einheitsquadraten kann man genau die folgenden zwölf Puzzlesteine herstellen, die paarweise verschieden sind:



Beim Pentapuzzle geht es darum, ein vorgegebenes Rechteck aus mindestens 60 Einheitsquadraten mit genau diesen zwölf Puzzlesteinen überschneidungsfrei auszulegen. Dabei dürfen die Puzzlesteine gedreht und gespiegelt, also beidseitig verwendet werden.

Aufgabe:
Gegeben ist ein Rechteck mit den Seiten a und b, a,b ganzzahlig und $a \times b \geq 60$.

1. Lassen sich die Puzzleteile überschneidungsfrei in dem Rechteck unterbringen, wenn gilt $a \times b = 60$? Erstellen Sie dazu ein Programm, das folgendes leistet:

   (a) Einlesen von a und b.

   (b) Bearbeitung des gegebenen Falles.

   (c) Feststellung, ob das Pentapuzzle für dieses Rechteck keine, genau eine oder mehrere Lösungen hat. Falls eine Lösung existiert, wird diese ausgegeben, beispielsweise unter Verwendung von zwölf verschiedenen Zeichen. Bei mehreren Lösungen werden maximal zehn verschiedene ausgegeben.

2. Stellen Sie fest, ob es Rechtecke gibt mit Seiten $a \geq 3$ und $b \geq 3$ und Fläche $a \times b > 60$, die nicht mit den zwölf Puzzlesteinen überschneidungsfrei belegt werden können.

3.3.2 Erläuterungen zur Aufgabe

Grundsätzlich läßt sich ein Programm, das der Aufgabenstellung genügt, leicht entwerfen. Ein einfaches Backtracking-Verfahren kann durch vollständiges Ausprobieren alle Lösungen für ein gegebenes Rechteck mit Flächeninhalt 60 finden; allerdings ist die Anzahl der Möglichkeiten, die Steine, die ja gedreht und gespiegelt werden dürfen, in das Feld zu legen, so groß, daß die Suche beschleunigt werden muß, um in vernünftiger Zeit Antworten zu finden.

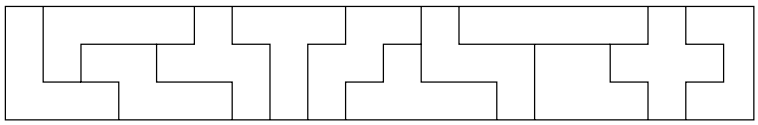


Abbildung 3.2: Eine Lösung für das 3×20-Feld.

Ganz offensichtlich kommt es nicht auf die Reihenfolge an, in der die Steine in das Feld gelegt werden. Man braucht also nicht alle Möglichkeiten, die Steine in das Feld zu legen, zu untersuchen, sondern kann sich auf die Fälle beschränken, die zu verschiedenen Ergebnissen führen. Dazu kann man eine bestimmte Reihenfolge der Belegung festlegen (beispielsweise oben beginnend zeilenweise von links nach rechts), oder man legt fest, daß der nächste anzulegende Stein an einen bereits gelegten Stein anschließen muß und nicht „frei“ liegen darf.

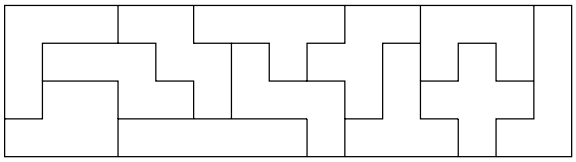


Abbildung 3.3: Eine Lösung für das 4×15-Feld.

Zur Beschleunigung ist es natürlich ebenfalls hilfreich, Abbruchkriterien zu formulieren, anhand derer möglichst früh entschieden werden kann, daß eine betrachtete Teillösung nicht mehr zu einer Komplettlösung ergänzt werden kann. Wird dann zusätzlich für die Steine eine Lege-Reihenfolge gewählt, die recht früh in Sackgassen führt (etwa diagonal, um möglichst früh getrennte Teilflächen entstehen zu lassen), kann eine große Anzahl vergeblicher Lösungsversuche eingespart und die Suche stark beschleunigt werden.

Die zweite Teilaufgabe sollte unter Berücksichtigung der Ergebnisse des ersten Aufgabenteils betrachtet werden. Für a und b kommen in der ersten Teilaufgabe nur die Wertepaare $(3, 20)$, $(4, 15)$, $(5, 12)$ und $(6, 10)$ in Frage, weil $a \times b = 60$ gelten soll. Die Paare $(1, 60)$ und $(2, 30)$ müssen nicht berücksichtigt werden, denn

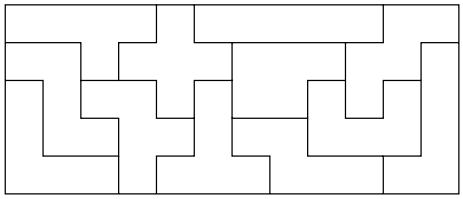


Abbildung 3.4: Eine Lösung für das 5×12-Feld.

die meisten Puzzlesteine brauchen eine Breite von drei Einheitsquadraten, um gelegt werden zu können. Rechtecke mit $a > b$ sind offensichtlich zu geeigneten Rechtecken mit $a < b$ symmetrisch und brauchen nicht eigens betrachtet zu werden. Unter Verwendung des für den ersten Aufgabenteil benötigten Programms erfährt man, daß es für diese vier Rechtecke jeweils Lösungen gibt. Konkrete Beispiele findet man in den Abbildungen 3.2 – 3.5.

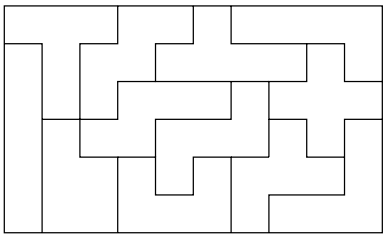


Abbildung 3.5: Eine Lösung für das 6×10-Feld.

Verlängert man eine oder gar beide Seiten eines dieser Rechtecke, so bleiben die gefundenen Lösungen natürlich gültig. Es genügt darum, ein 7×9- und ein 8×8-Feld zu untersuchen. Weil man zu diesen beiden Feldern Lösungen finden kann (bei denen natürlich 3 bzw. 4 Einheitsquadrate frei bleiben, Beispiele für Lösungen sieht man in den Abbildungen 3.6 und 3.7), gibt es für alle Rechtecke mit Seiten $a \geq 3$ und $b \geq 3$, bei denen $a \times b > 60$ gilt, überschneidungsfreie Belegungen mit den zwölf Puzzlesteinen.

3.3.3 Grobstruktur eines Programms

Es gibt insgesamt zwölf Puzzlesteine, die gedreht und gespiegelt werden dürfen, wobei allerdings nicht durch jede Drehung oder Spiegelung eine andere Figur entsteht. Insgesamt gibt es 63 verschiedene Steine. Diese sollte man jedoch nicht in jedem Backtrack-Schritt neu aus den zwölf Urbildern durch Drehen und Spiegeln gewinnen; effizienter ist es, dies einmalig beim Programmstart zu tun oder

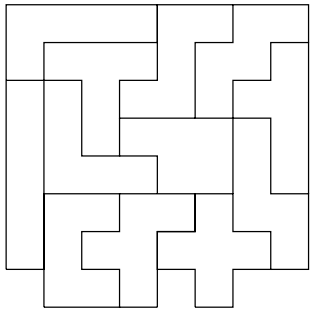


Abbildung 3.6: Eine Lösung für das 8×8-Feld.

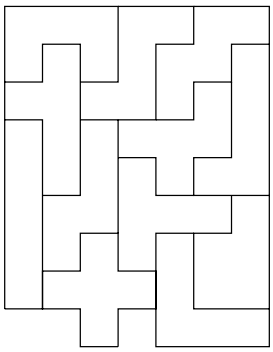


Abbildung 3.7: Eine Lösung für das 7×9-Feld.

gleich alle 63 Steine als Konstanten im Programm zu vereinbaren. Es bietet sich an, zur Speicherung eines Steins ein Array der Länge 4 zu verwenden und die relativen Koordinaten der Einheitsquadrate zu einem bestimmten Einheitsquadrat abzuspeichern. Für dieses „Basisquadrat“ werden implizit die Koordinaten $(0,0)$ angenommen. Dann kann man zum Beispiel den ersten Puzzlestein durch die Koordinatenfolge $(-1,1)$, $(0,1)$, $(1,1)$, $(0,2)$ darstellen.

Das zu belegende Feld sollte als zweidimensionales Array realisiert werden. Wenn man das Feld um acht Zeilen und Spalten größer als nötig macht und ringsum die äußeren vier Zeilen und Spalten als konstant belegt kennzeichnet, kann man sich beim Setzen der Steine die besondere Behandlung der Randfälle sparen. Es genügt dann, für jeden Stein zu prüfen, ob das benötigte Einheitsquadrat im Feld noch frei ist. Die Koordinaten der zu belegenden Felder ergeben sich einfach durch Addition der relativen Koordinaten der Einheitsquadrate des Puzzlesteins zu den Koordinaten des aktuellen Startfeldes.

Zu jedem der 63 verschiedenen Steine muß bekannt sein, aus welchem Urbild er hervorgegangen ist; im Laufe der Suche nach einer Lösung müssen die benutzten Urbilder gekennzeichnet werden, damit kein Stein mehrfach verwendet wird. Dafür kann am leichtesten ein Array der Länge 12 benutzt werden. Darin wird vermerkt, Steine zu welchen Urbildern bereits benutzt wurden.

Man kann unter Benutzung dieser einfachen Datenstrukturen in einem Backtracking-Verfahren beginnen, die Steine anzulegen; dabei soll die gewählte Legerichtung und die Bedingung, daß Steine immer ohne unnötige Zwischenräume an schon liegende Steine angelegt werden sollen, beachtet werden. Entsteht durch das Legen des Steins innerhalb des Rands eine abgeschlossene Fläche, deren Größe nicht ohne Rest durch fünf teilbar ist, so kann der zuletzt gelegte Stein wieder aufgenommen und die nächste Möglichkeit probiert werden. Ist der letzte Stein verwendet worden, kann die gefundene Lösung ausgegeben werden.

Auf diese Art können auch Rechtecke behandelt werden, die mehr als 60 Einheitsquadrate groß sind; das ermöglicht es, das Programm auch zur Untersuchung der für den zweiten Aufgabenteil relevanten Größen 7×9 und 8×8 einzusetzen. Es ist allerdings mit diesem Verfahren im allgemeinen nicht so, daß alle möglichen Lösungen für "zu große" Felder gefunden werden: Durch das Anlegen der neuen Steine an vorher gelegte Steine werden ausschließlich zusammenhängende Belegungen gefunden. Trotz dieser Einschränkung bleibt das Programm für die zwei relevanten Sonderfälle nutzbar, sofern man nicht mehr prüft, ob abgeschlossene Flächen entstanden sind, die nicht lückenlos ausfüllbar sind.

3.3.4 Halbformale Programmbeschreibung

```
Pentapuzzle:
  lies a und b ein;
  wenn a > b, dann vertausche a und b;
  initialisiere Feld der Größe (a + 8) × (b + 8) als leer,
    Ränder als belegt;
  Lösungszähler := 0;
  Löse(0, Lösungszähler);
  gib Ergebnis aus.

Löse(Tiefe, Zähler):
  wenn Tiefe = 12, gib Lösung aus und erhöhe Zähler;
  sonst:
    wenn < 10 Lösungen gefunden:
      suche die erste freie Position im Feld von oben links;
      für alle 12 Steine:
```

```
        wenn Stein noch nicht gelegt:
          für alle Anlegemöglichkeiten dieses Steins:
            für alle spezifischen Drehungen und Spiegelungen:
              berechne Koordinaten;
              wenn Stein paßt:
                lege Stein;
                Löse(Tiefe+1, Zähler);      (rekursiv)
                nimm Stein zurück;          (Backtracking)
end. {Löse}
```

3.4 Kuriositäten und besonders nette Formulierungen

- Dem hier verwandten Verfahren sind vielfältige Grenzen gesetzt, von denen hier einige aufgezeigt werden sollen.
- Darüber streiten sich die Geister der OCR-Experten.
  wann? Nachts zwischen 12 und 1?
- Die erste freie Zeile, die nicht ausgefüllt ist.
- Daß bisher noch keine Lösung gefunden wurde, wird dem Computer auch noch klar gemacht. *(Im Klartext: Initialisiere Anzahl der Lösungen mit 0)*
- Die Steine des Pentapuzzles verbindet, daß sie alle aus 5 Einheitsquadraten bestehen, deren zwei sich mindestens entlang einer Seite berühren müssen.
- Steinprotokoll
- optimaler
- Das Erkennen und die Ausgabe nimmt keinerlei Zeit in Anspruch.
- Wenn Sie diese Aufgabe in den Händen halten, liegt der Gedanke nahe, daß es sich bei der Lösung dieser Aufgabe wohl um einen echten Overkill handelt.
  Ausarbeitung Aufgabe 1: 128 Seiten
- Der Grundgedanke zur Lösung dieser Aufgabe war recht einfach, das Problem lag im Detail.
- Die Idee basiert auf dem Gedanken, daß wenn ich weiß, wie z. B. eine Banane ungefähr aussieht, trotzdem andere Bananen erkenne, auch wenn sie größer oder kleiner, runder oder gestreckter sind. Das Grundprinzip ist: Ich weiß, eine Banane ist eigentlich gelb, ist leicht gebogen und hat an einer Seite einen verrindeten Ansatz. Wenn ich jetzt noch die ungefähre räumliche Größenordnung kennenlerne, dann erkenne ich eine Banane.
- Nebenbei: Bei einem Quattropuzzle (4er-Puzzle) gibt es bei einer Feldgröße von 5×5 genau 1000 Lösungen weniger, wie *(ALS!!!)* in der ersten Runde beim "Haus des Nikolaus", nämlich genau 12760! ZUFALL!

4 13. Wettbewerb: Die Aufgaben der 1. Runde

4.1 Schiffe versenken

4.1.1 Aufgabenstellung

Schiffe versenken wird von zwei Spielern gespielt. Zu Beginn des Spiels positioniert jeder auf seinem eigenen Spielfeld Schiffe, indem er einmal 2, zweimal 3, einmal 4 und einmal 5 in einer Reihe oder einer Spalte liegende Kästchen markiert. Natürlich dürfen sich die Schiffe nicht überkreuzen und nicht über das Spielfeld hinausragen. Sonst ist alles erlaubt. Im Verlauf des Spiels raten beide Spieler abwechselnd die Positionen der gegnerischen Schiffe. Wird ein Schiff getroffen, so muß der Spieler einen "Treffer" melden und die Position markieren. Sobald alle Positionen eines Schiffes getroffen sind, ist es "versenkt" zu melden. Ziel des Spiels ist, alle Schiffe des Gegners zu versenken, Sieger ist der Spieler, dem das zuerst gelingt.

Aufgabe:
Schreibe ein Programm, das mit Dir "Schiffe versenken" spielt, indem es ein 10×10-Spielfeld darstellt, Schiffe positioniert (ohne die Positionen anzuzeigen) und Deine Rateversuche entgegennimmt, anzeigt und beantwortet.

4.1.2 Lösungsidee

Zunächst werden die fünf Schiffe auf dem 10×10-Spielfeld versteckt. Dies muß zufällig geschehen, denn stets gleiche Schiffspositionen sind für die Spieler keine Herausforderung. Beim Positionieren der Schiffe ist darauf zu achten, daß die geforderten Bedingungen eingehalten werden, d. h. daß die Schiffe sich nicht überschneiden und nicht über den Spielfeldrand hinausragen. Ein Schiff kann nur mehrere Kästchen in einer Reihe oder in einer Spalte belegen, diagonale Positionierung ist nicht erlaubt. Die Regel, daß Schiffe sich nicht berühren dürfen, ist zwar weit verbreitet, aber sie ist in der Aufgabenstellung nicht angegeben, so daß sie hier auch nicht berücksichtigt werden muß.

Nach der Positionierung der Schiffe wird das zunächst völlig leere Spielfeld auf dem Bildschirm angezeigt, anschließend werden die Eingaben des Spielers entgegengenommen. Dabei werden Eingaben außerhalb des Spielfeldes oder auf ein schon zuvor geratenes Feld abgefangen. Geratene Felder werden unterschiedlich markiert, je nachdem, ob es sich um einen Schuß ins Wasser oder einen Treffer handelt. Ein versenktes Schiff kann auf dem Spielfeld anders angezeigt werden als ein angeschossenes, das ist aber natürlich nicht zwingend notwendig und wird von

unserem Programm auch nicht gemacht. (Ein nettes Zeichen für Wasser ist ~.) Außerdem muß festgehalten werden, wie viele Schiffe schon versenkt sind, damit der Computer das Spiel mit einer entsprechenden Mitteilung beenden kann, wenn das letzte Schiff versenkt wurde.

Für das Spielfeld *TSpielfeld* wird ein Array[1..10,1..10] verwendet. Jede Zelle des Feldes kann die Zustände *wasser, treffer, keinTreffer, zweier, dreier1, dreier2, vierer* oder *fuenfer* annehmen. Außerdem sind alle Schiffe in einem eindimensionalen Feld *flotte* gespeichert, in dem zu jedem Schiff alle von ihm belegten Koordinaten abgelegt sind. Wird ein Schiff *schiff* getroffen, so läßt sich mit *flotte*[*schiff*] = leer überprüfen, ob dieses Schiff nun versenkt wurde. Alle noch nicht versenkten Schiffe sind in *aktSchiffe* gespeichert. Erst wenn *aktschiffe* leer ist, wurden alle Schiffe versenkt und das Spiel ist beendet.

4.1.3 Programmablaufprotokoll

Das Spielfeld wird folgendermaßen dargestellt:

```
  Schiffe versenken

   12345678910
1  ~~~~~~~~~~
2  ~~~~~~~~~~
3  ~~~~~~~~~~
4  ~~~~~~~~~~
5  ~~~~~~~~~~
6  ~~~~~~~~~~
7  ~~~~~~~~~~
8  ~~~~~~~~~~
9  ~~~~~~~~~~
10 ~~~~~~~~~~

   Eingabe der x-Koordinate:
   Eingabe der y-Koordinate:
```

Je nach Zielgenauigkeit erscheint eine der Ausgaben `daneben - leider kein Treffer`, `Treffer!` oder `Treffer! Versenkt!` Außerdem werden natürlich im Spielfeld die beschossenen Felder eingetragen, entweder als `X` für einen Treffer oder als `o` für daneben.

Nach einiger Zeit kann das Spielfeld also beispielsweise so aussehen:

```
  Schiffe versenken

   12345678910
1  ~~~~~~~~~~
2  ~~~~~~~o~~
3  ~~~~~~~~~~
4  ~~~~~~~~~~
5  ~~~X~~~~~~
6  ~~oX~~o~~~
7  ~~~X~~~~~~
8  ~~~o~~~XXo
9  ~~~~~~~~~~
10 ~~~~~~~~o~

   Eingabe der x-Koordinate: 4
   Eingabe der y-Koordinate: 5

 Treffer! Versenkt!
```

4.1.4 Programmtext

```
program SchiffeVersenken;

uses crt;

type Tzustand = (treffer,wasser,keinTreffer,zweier,dreier1,dreier2,
                 vierer,fuenfer); { mögl. Belegung einer Koordinate }

const bezeichnung : array[zweier..fuenfer] of string =
                 ('Zweier','Dreier','Dreier','Vierer','Fuenfer');
                                          { Hilfsfeld für die Ausgabe }
      bootslaenge : array[zweier..fuenfer] of byte =
                 (2,3,3,4,5);                              { Hilfsfeld }
      feldzeichen : array[treffer..keinTreffer] of char =
                 ('X','~','o');   { Zeichen für die Spielfeldausgabe }

type Tspielfeld = array[1..10,1..10] of Tzustand;
     Tposition = byte;                                  { verschlüsselt! }
     Tschiff = set of Tposition;          { ein Schiff = mehrere Positionen }
     Tflotte = array[zweier..fuenfer] of Tschiff;
     Tmarkschiff = set of Tzustand;          { welches Schiff ist noch aktiv }
```

```
var spielfeld : Tspielfeld;                                  { 10*10 Feld }
      flotte : Tflotte;                                  { Positionen aller Schiffe }
      aktSchiffe : Tmarkschiff;                              { alle aktiven Schiffe }
      x,y,code : byte;
      schiff : Tzustand;

procedure initialisierung(var spielfeld: Tspielfeld; var flotte : Tflotte;
                          var aktSchiffe: Tmarkschiff);
{ Setzt Wasser (=unbekannt) im ganzen Spielfeld. }
{ Zu Anfang sind alle Schiffe aktiv. }
{ Die Positionen der Schiffe werden später bestimmt. }
var x,y : byte;
    laufschiff : Tzustand;
begin
  for x := 1 to 10 do
    for y := 1 to 10 do
      spielfeld[x,y] := wasser;
  for laufschiff := zweier to fuenfer do
    flotte[laufschiff] := [];
  aktSchiffe := [zweier..fuenfer]
end;  { initialisierung }

procedure positioniereSchiffe(var spielfeld : Tspielfeld;
                              var flotte : Tflotte);

function ragtNichtHinaus(x,y : byte; richtung, laenge : byte): boolean;
begin
  if richtung = 0 then                                  { senkrecht nach unten }
    ragtNichtHinaus := (y+laenge-1) <= 10
  else ragtNichtHinaus := (x+laenge-1) <= 10
end;  { ragtNichtHinaus }

function ueberkreuztNicht(spielfeld : Tspielfeld; x,y, richtung, laenge :
                          byte): boolean;
{ zwei Schiffe dürfen keine gemeinsame Koordinate haben }
var ueberschneidung : boolean;
begin
  ueberschneidung := false;
  while (laenge <> 0) and not (ueberschneidung) do
  begin
    if spielfeld[x,y] in [zweier..fuenfer] then
```

```
      ueberschneidung := true
    else begin
      dec(laenge);
      if richtung = 0 then
        inc(y)
      else inc(x)
    end
  end;  { while (laenge <> 0) }
  ueberkreuztNicht := not (ueberschneidung)
end;  { ueberkreuztNicht }

procedure schiffEndgueltigSetzen(var spielfeld : Tspielfeld; x,y,richtung,
                                     laenge : byte;
                                     schiff : Tzustand; var flotte: Tflotte);
{ Spielfeld wird z.T. mit Schiffdaten belegt }
begin
  while (laenge <> 0) do
  begin
    spielfeld[x,y] := schiff;
    flotte[schiff] := flotte[schiff] + [x*10+y];
    dec(laenge);
    if richtung = 0 then
      inc(y)
    else inc(x)
  end
end;  { schiffEndgueltigSetzen }

var laufschiff : Tzustand;
    x,y,richtung : byte;
begin
  for laufschiff := zweier to fuenfer do
  begin
    repeat
      x := random(10) + 1;                          { linke obere Ecke }
      y := random(10) + 1;                          { wählen           }
      richtung := random(2);                { 0 = vertikal; 1 = horizontal }
    until ragtNichtHinaus(x,y,richtung,bootslaenge[laufschiff]) and
      ueberkreuztNicht(spielfeld,x,y,richtung,bootslaenge[laufschiff]);
    schiffEndgueltigSetzen(spielfeld,x,y,richtung,bootslaenge[laufschiff],
                                   laufschiff,flotte)
  end
end;  { positioniereSchiffe }
```

```
procedure rateversuch(var x,y, code : byte);
  { Spieler wird abgefragt }
begin
  gotoxy(5,17);
  write('Eingabe␣der␣x-Koordinate:␣');
  readln(x);
  gotoxy(5,18);
  write('Eingabe␣der␣y-Koordinate:␣');
  readln(y);
  code := x * 10 + y
end;  { rateversuch }

procedure darstellen(spielfeld : Tspielfeld);
  { grafische Darstellung des Spielfeldes auf dem Bildschirm }
var laufx, laufy : byte;
begin
  gotoxy(5,3);
  writeln('Schiffe␣versenken');
  gotoxy(6,5);

  for laufx := 0 to 10 do
    for laufy := 0 to 10 do
    begin
      if (laufx = 0) and not(laufy = 0) then
        write(laufy)
      else if (laufy = 0) and not(laufx = 0) then
      begin
        gotoxy(3,5+laufx);
        write(laufx)
      end
      else if not(laufx = 0) and not(laufy = 0) then
      begin
        if spielfeld[laufx,laufy] in [zweier..fuenfer] then
          spielfeld[laufx,laufy] := wasser;
        gotoxy(5+laufx,5+laufy);
        write(feldzeichen[spielfeld[laufx,laufy]])
      end
    end  { for laufx}
end;  { darstellen }
```

```
begin  { schiffeVersenken }
  randomize;
  initialisierung(spielfeld,flotte,aktSchiffe);
  positioniereSchiffe(spielfeld,flotte);
  repeat
    clrscr;
    darstellen(spielfeld);
    rateversuch(x,y,code);
    if spielfeld[x,y] in [zweier..fuenfer] then
    begin
      schiff := spielfeld[x,y];
      spielfeld[x,y] := treffer;
      gotoxy(5,20);
      writeln('Treffer!');
      flotte[schiff] := flotte[schiff] - [code];
      if flotte[schiff] = [] then
      begin
        aktSchiffe := aktSchiffe - [schiff];
        gotoxy(15,20);
        writeln('Versenkt!')
      end
    end  { if spielfeld[x,y] }
    else begin
      spielfeld[x,y] := keinTreffer;
      gotoxy(5,20);
      writeln('daneben␣-␣leider␣kein␣Treffer')
    end;
    darstellen(spielfeld);
    readln
  until aktSchiffe = [];
  gotoxy(5,22);
  writeln('Das␣Spiel␣ist␣beendet!␣Alle␣Schiffe␣sind␣versenkt!');
  readln
end.  { schiffeVersenken }
```

4.2 Nochmal Schiffe versenken

4.2.1 Aufgabenstellung

Beschreibe eine Strategie, um mit möglichst wenigen Rateversuchen alle gegnerischen Schiffe zu versenken. Wieviele Rateversuche benötigt Deine Strategie durchschnittlich? Wieviele Rateversuche benötigt sie höchstens? Welche Schiffspositionierungen sind für Deine Strategie besonders ungünstig? Welche sind besonders günstig?

4.2.2 Lösungsidee

Eine einfache Strategie ist, das 10×10-Feld schachbrettartig zu beschießen, also jedes 2. Kästchen und das reihenweise versetzt. Mit dieser Strategie benötigt man maximal 71 Versuche, bis alle Schiffe versenkt sind. Dieser schlechteste Fall tritt ein, wenn man mit dem Beschuß auf A2 beginnt, der Zweier waagrecht im rechten unteren Eck liegt (Kästchen J9 und J10), man nach Schema auf J9 schießt und dann zunächst I9 und J8 prüft, bevor man das Schiff versenkt, und man außerdem für jedes übrige Schiff nach dem ersten Treffer jeweils die maximale Zahl Fehlschüsse getan hat (das sind jeweils drei). Damit ergeben sich 50 Schüsse für das Raster + 5 zusätzliche Schüsse für den Fünfer (2 Kästchen liegen nicht im Raster plus 3 Fehlschüsse oder 3 Felder liegen nicht im Raster, dann gibt es höchstens noch 2 Fehlschüsse) + 5 zusätzliche Schüsse für den Vierer + $2 \cdot 4$ für die Dreier + 3 für den Zweier.

Die angegebene Strategie läßt sich noch verfeinern, indem man beispielsweise auf ein gröberes Raster umsteigt, wenn der Zweier gefunden wurde, oder indem man in der Mitte des Spielfeldes statt am Rand beginnt.

Im Mittel werden mit der Zweierstrategie zwischen 45 und 50 Versuche benötigt.

Wie bereits erwähnt ist es besonders ungünstig, wenn man den letzten Zweier erst ganz zum Schluß erwischt, also das Raster komplett beschießen muß. Günstiger ist es, wenn die Schiffe alle dicht beisammen liegen, und zwar in dem Teil des Spielfeldes, in dem man mit dem Beschuß anfängt.

4.3 Rätselecke

4.3.1 Aufgabenstellung

In einem Geschichtsrätsel werden die folgenden Daten der Rechentechnik gefragt:

1. Geburt von Blaise Pascal
2. Der erste Mikroprozessor
3. Modell der Turing-Maschine
4. Babbages Differenziermaschine
5. Einführung der modernen Lochkartentechnik
6. Erfindung der Leibnizschen Rechenmaschine
7. Erfindung des Abakus

Da es mehr auf die geschichtlichen Zusammenhänge als auf die genauen Daten ankommt, sollen die Ereignisse nur in die richtige chronologische Reihenfolge gesetzt werden. Die Bewertung des Rätsels ergibt sich aus der längsten (nicht notwendig ununterbrochenen) Kette richtig geordneter Ereignisse.

Beispiele:
Die Jahreszahlen
1) 1725 2) 1431 3) 1925 4) 1615
führen zur Folge 2 4 1 3.

Die richtige Reihenfolge sei 1 2 3 4 5 6 7. Die Reihe 1 2 3 4 6 5 7 ergibt dann 6 Punkte für 1 2 3 4 5 7, das Ereignis 6 ist falsch eingeordnet. Die Reihe 3 4 5 6 1 7 2 ergibt 5 Punkte für 3 4 5 6 7, die Ereignisse 1 und 2 sind falsch eingeordnet.

Aufgabe:
Bringe zunächst die Daten in die richtige chronologische Reihenfolge. Schreibe ein Programm, welches zuerst die Zahlenfolge für die richtige Lösung einliest und dann beliebig viele Zahlenfolgen für Rateversuche. Zu jeder geratenen Zahlenfolge gibt es die Punktzahl aus. Gib Deinem Programm die richtige Reihenfolge der obigen Ereignisse aus der Geschichte der Rechentechnik als erste Eingabe. Welche Punktzahlen erreichen die folgenden Rateversuche?

a) 2 3 5 4 6 1 7

b) 7 1 6 4 5 3 2

c) 1 2 3 4 5 6 7

4.3.2 Lösungsidee

Die richtige Folge *okfolge* wird mittels einer Funktion f auf die Folge 1 2 3 4 5 6 7 abgebildet. Für die Folge 7 1 6 5 3 2 4 gilt also beispielsweise $f(7) = 1$, $f(1) = 2$, $f(6) = 3$ usw. (Dies ist viel einfacher als es klingt, da man nur statt eines Elementes seinen Index in der Folge betrachten muß.) Auf die geratene Folge *testfolge* wird die gleiche Funktion f angewendet. In der so erhaltenen Folge wird nun die längste monoton steigende Teilfolge gesucht. Diese Teilfolge muß nicht zusammenhängend sein. Zum Beispiel ist in der Folge 4 2 7 5 6 1 3 die gesuchte Teilfolge 2 5 6. Eine (ineffiziente, aber für kleine Folgen praktikable) Methode, die gesuchte Teilfolge zu finden, ist, systematisch alle möglichen Teilfolgen zu erzeugen, sie zu prüfen, ob sie aufsteigend sind, und sich jeweils die Länge der längsten bisher gefundenen aufsteigendenTeilfolge zu merken. Alle Teilfolgen kann man folgendermaßen erzeugen: Sei n (im Programm *maxAnzahl*) die Länge der Gesamtfolge (hier 7). Dann gibt es $2^n - 1$ nicht notwendig zusammenhängende Teilfolgen (entsprechend der Anzahl der nichtleeren Teilmengen einer n-elementigen Menge). Jeder Teilfolge ordnet man nun die Binärdarstellung einer Zahl zwischen 1 und $2^n - 1$ zu, indem man genau an den Positionen in der Binärdarstellung Einsen setzt, die den Positionen der Zahlen in der Teilfolge entsprechen. Man durchläuft nun alle Binärdarstellungen und betrachtet in der Testfolge genau die Elemente, die den Einsen in der Binärdarstellung entsprechen. Diese Elemente werden überprüft, ob sie eine aufsteigende Folge ergeben. Im angegebenen Programm wird nicht einmal dies geprüft, sondern nur die Länge der längsten aufsteigenden Teilfolge innerhalb dieser Folge berechnet.

Beispiel: *binarray* 1 0 0 0 1 0 1, *testfolge* 1 3 4 7 6 5 2, betrachtet werden 1, 6 und 2. Hierin hat die längste aufsteigende Folge die Länge 2. Die längste so ermittelte Teilfolge (bzw. eine längste Teilfolge, wenn es mehrere gleichlange Teilfolgen gibt) ist gerade die gesuchte und ihre Länge entspricht der Punktzahl, die mit der Testfolge erzielt wurde.

Die richtige Reihenfolge für das angegebene Beispiel ist 7 1 6 4 5 3 2. Daraus folgen für a) 1 Punkt, b) 7 Punkte und c) 3 Punkte.

4.3.3 Halbformale Programmbeschreibung

```
lies okfolge ein;
wiederhole:
  lies testfolge ein;
  maxlaenge := 0;
  wende f auf testfolge an;
  { f : Indizes der Elemente der okfolge }
```

```
  speichere so erhaltene Folge in testfolge;
  für i := 1 bis 2n − 1:
    schreibe i in Binärdarstellung in binarray;
    betrachte nur Elemente aus testfolge, die an gleicher Position
      wie die Einsen in binarray stehen;
    berechne für diese Elemente längste aufsteigende Teilfolge;
    aktmax := Länge dieser Teilfolge;
    wenn aktlaenge > maxlaenge:
      aktualisiere maxlaenge;
    gib maxlaenge aus
  bis keine weiteren Versuche gewünscht;
gib okfolge aus.
```

4.3.4 Programmtext

```
program Raetselecke;
{ Es werden keinerlei Fehlerabfragen gemacht, }
{ falsche Eingaben sind also möglich!          }
const
  maxAnzahl = 7;
var
  okfolge, testfolge : array[1..maxAnzahl] of integer;
  laenge, jj : integer;
  weiter : char;

procedure liesOkFolge;
var i, zahl: integer;
begin
  for i := 1 to maxAnzahl do
  begin
    write ('Bitte␣die␣',i,'.␣Zahl␣der␣richtigen␣Folge:␣');
    readln(zahl);
    okfolge[zahl] := i
  end;
  writeln
end;  { liesOkFolge }

procedure liesTestFolge;
var i, j, zahl: integer;
begin
  for i := 1 to maxAnzahl do
```

```
  begin
    write ('Bitte␣den␣',i,'.␣Testkandidaten:␣');
    readln(zahl);
    j := 1;
    while okfolge[j] <> zahl do
      j := j + 1;
    testfolge[i] := j
  end;
  writeln
end;  { liesTestFolge }


function berechneLaenge : integer;
var
  binarray: array[1..maxAnzahl] of integer;
  i, j, aktmax, aktlaenge, maxlaenge, hilfmax, ii: integer;
begin
  hilfmax := 1;
  for i := 1 to maxAnzahl do                    { berechne 2 hoch maxAnzahl }
    hilfmax := hilfmax * 2;
  hilfmax := hilfmax - 1;
  maxlaenge := 0;
  for i := 1 to hilfmax do
  begin
    ii := i;
    for j := maxAnzahl downto 1 do                          { fülle binarray }
    begin
      binarray[j] := ii mod 2;
      ii := ii div 2
    end;
    aktmax := 0;
    aktlaenge := 0;
    for j := 1 to maxAnzahl do
      if binarray[j] = 1 then
        if testfolge[j] > aktmax then
        begin
          aktmax := testfolge[j];
          aktlaenge := aktlaenge + 1
        end;
      if aktlaenge > maxlaenge then
        maxlaenge := aktlaenge
  end;
```

```
  berechneLaenge := maxlaenge
end;  { okCount }

begin                                                    { Hauptprogramm }
  liesOkFolge;
  weiter := 'j';
  while (weiter = 'j') or (weiter = 'J') do
  begin
    liesTestFolge;
    laenge := berechneLaenge;
    writeln ('Die␣Punktzahl␣ist:␣',laenge);
    writeln;
    write ('Neuer␣Versuch?␣(j/n):␣');
    readln(weiter);
    writeln
  end;
  writeln;
  write('Die␣richtige␣Reihenfolge␣ist␣␣␣');
  for jj := 1 to maxAnzahl do
    write(okfolge[jj],'␣␣');
  writeln;
  readln
end.
```

4.4 Verschlüsselung

4.4.1 Aufgabenstellung

Sabine und Marcella schreiben sich seit geraumer Zeit Briefe in deutscher Sprache per elektronischer Post (e-mail) von Hannover nach Wien. Da die e-mail für die Mitschülerinnen und -schüler ebenfalls einzusehen ist, vereinbaren sie eine Verschlüsselung. Es können nur Buchstaben, Ziffern und die Sonderzeichen ? ! § $ % & [ ] # * dazu benutzt werden. Z. B. verschlüsseln sie durch Verschiebung: Verschiebung um 3 heißt, aus a wird d, aus b wird e usw.

- Überlege Dir zwei andere, nicht auf Verschiebung basierende Verfahren, Text zu verschlüsseln. Beschreibe jeweils das Verfahren und die Informationen, die die Empfängerin braucht, um den Text zu entschlüsseln. Gib an, welche Vor- und Nachteile die Verfahren haben. Verschlüssele mit beiden Verfahren den folgenden Satz: Wenn vor Fliegen Fliegen fliegen, fliegen Fliegen Fliegen hinterher.
- Finde eine Möglichkeit, den Text so zu verschlüsseln, daß der Ergebnistext kürzer ist als das Original. (Erklärung!)

4.4.2 Lösungsidee

Im *verschlüsselten* Text dürfen nur die in der Aufgabenstellung angegebenen Zeichen vorkommen; damit ist aber noch nichts über die im Originaltext vorkommenden Zeichen gesagt. Insbesondere können im Originaltext Leerzeichen, Kommata und Punkte stehen, wie man schon am Fliegensatz sieht. Diese müssen also mitverschlüsselt werden. Aus dem Fliegensatz geht auch hervor, daß Groß- und Kleinbuchstaben unterschieden werden müssen, denn sonst ist keine korrekte Entschlüsselung möglich.

Eine recht einfache Möglichkeit, Text zu verschlüsseln, ist beispielsweise, mit Hilfe einer Tabelle jedes Zeichen auf ein anderes Zeichen abzubilden. Etwa $a \to F$, $b \to$ #, $c \to$], $A \to z$, $B \to$!, . $\to X$ usw.

Ein solches Verfahren ist nicht so leicht zu entschlüsseln wie eine reine Verschiebung; sehr schwierig ist die Entschlüsselung aber auch nicht, da man aus der Häufigkeit der Zeichen auf ihre Bedeutung schließen kann. Klarerweise braucht hier die Empfängerin die Tabelle, nach der verschlüsselt wurde.

Wenn man im Deutschen häufig auftretende Silben oder Buchstabenkombinationen durch einzelne Zeichen verschlüsselt, verkürzt das angegebene Verfahren in

der Regel auch. Beispielsweise $sch \rightarrow Y$, $gen \rightarrow$ §, $er \rightarrow a$ usw. Dies erschwert die Entschlüsselung zusätzlich.

Ein anderes Verschlüsselungsverfahren, das den Text allerdings verlängert, ist folgendes: Vor (oder auch nach) jedem Zeichen steht eine Zahl. Diese gibt an, um wieviele Stellen im Alphabet das Zeichen verschoben werden muß, um das Originalzeichen zu finden. Dabei könnte man noch wechseln zwischen Verschiebung nach rechts und Verschiebung nach links. Es muß natürlich klar sein, an welcher Stelle und in welcher Reihenfolge die Sonderzeichen stehen (beispielsweise wahllos in das Alphabet eingestreut). Außerdem müssen Zahlen, die nicht für die Verschiebung stehen, sondern ein verschlüsseltes Zeichen darstellen, besonders kenntlich gemacht werden.

Wir legen beispielsweise die folgende Reihenfolge fest:
A b C D e f G H I j k l M N O P q r s t U V W X Y z 0 ? 1 ! 2 § 3 $ 4 % 5 & 6 [7] 8 # 9 . Leerzeichen , : ; a B c d E F g h i J K L m n o p Q R S T U v w x y Z
Als Sonderzeichen für eine Ziffer verwenden wir $*$.
Aus "Informatik 13" könnte dann bei alternierender Verschiebung (beginnend mit rechts) werden:

3f 15C 0f 3R 6l 2o 6*9 5Y 7B 23$ 7[1! 5?

(Die Leerzeichen wurden nur zur besseren Lesbarkeit eingefügt.)

4.5 Rotieren nach Plan

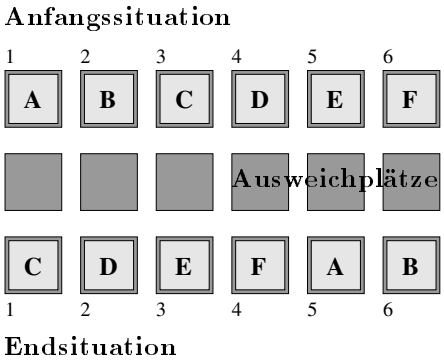
4.5.1 Aufgabenstellung

Du bist wegen Deiner Schnelligkeit und Geschicklichkeit bereits unter den besten Spielerinnen und Spielern des Wettkampfs. Erst im letzten Spiel wird die Entscheidung über die Siegerin oder den Sieger fallen. Folgende Aufgabe ist zu lösen:

Vor jedem Spieler stehen auf n umrandeten Plätzen riesige Schaumstoffwürfel in einer Reihe. Jeder Würfel trägt eine eindeutige Kennung. Die Spieler sollen ihre Würfel um i Positionen nach links rotieren. Nachher sollen alle Würfel wieder auf den umrandeten Plätzen stehen. Jedem Spieler stehen noch n umrandete Ausweichplätze zur Verfügung, die er während des Umgruppierens benutzen kann, ja sogar benutzen muß, denn ein Spieler kann immer nur einen Würfel gleichzeitig tragen.

Sieger wird, wer die wenigsten Ausweichplätze benutzt und wer bei gleicher Anzahl benutzter Ausweichplätze die wenigsten Würfeltransporte benötigt. Die Größen n und i werden erst unmittelbar vor dem Spiel den Spielern gesagt.

Beispiel mit 6 Würfeln (A..F), die um zwei Positionen nach links rotieren sollen, und mit 6 Ausweichplätzen:



Eine Möglichkeit wäre, erst alle n Würfel nacheinander auf die n Ausweichplätze zu tragen und sie dann von dort aus auf die gewünschten Plätze zu bringen. Aber das wäre wohl keine gute Lösung.

Also mußt Du Dir einen Plan zurechtlegen.

Beschreibe Deinen Plan und mache deutlich, daß Du damit die Spielaufgabe lösen kannst. Kannst Du mit Deinem Plan auch Sieger werden? Schätze ab, wieviel

Ausweichplätze Du benötigst und wie oft Du Schaumstoffwürfel transportieren mußt.

4.5.2 Lösungsidee

Die mehrfache Benutzung des gleichen Ausweichplatzes zählt als nur ein benutzter Ausweichplatz. Gefragt ist also nach der Minimierung der *gleichzeitig* benutzten Ausweichplätze. Die Antwort vorweg: Es genügt immer *ein* Ausweichplatz.

Die Anzahl der notwendigen Würfeltransporte beträgt $n + \text{ggt}(n, i)$. Wir betrachten nur solche i, die kleiner als n sind; für größere i gilt $i_{neu} := i_{alt} \bmod n$. Wenn n und i teilerfremd sind ($\text{ggt}(n, i) = 1$), setzt man Würfel A auf einen Ausweichplatz, auf den freigewordenen Platz setzt man den Würfel i Positionen rechts vom freigewordenen Platz. Mit dem nun freigewordenen Platz verfährt man genauso (wenn es rechts von dort keine i Würfel mehr gibt, zählt man von links her weiter), bis der Platz für Würfel A freigeworden ist. Bis jetzt hat man n Transporte durchgeführt (A auf Ausweichplatz und $n-1$ Würfel um i Positionen rotiert). A wird nun auf seinen endgültigen Platz gesetzt, so daß insgesamt $n+1$ Transporte durchgeführt wurden.

Sind n und i nicht teilerfremd, funktioniert das beschriebene Verfahren nur teilweise, denn dann ist A nicht der n-te, sondern der n/i-te Würfel, dessen endgültiger Platz frei wird. Also wird A dorthin gesetzt und ein anderer Würfel, der sich noch nicht in seiner Endposition befindet, muß ausweichen. Dies führt dann zu der angegebenen Zahl von $n + \text{ggt}(n, i)$ Würfeltransporten.

Es gibt noch andere Möglichkeiten, die Würfel zu rotieren, aber keine benötigt weniger Würfeltransporte.

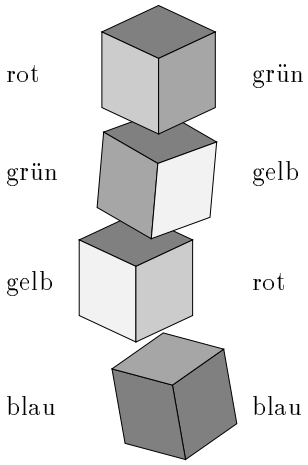
Beispiel 1: Sei $n = 5$, $i = 3$, d. h. die Würfel A, B, C, D und E stehen auf den Feldern 1, 2, 3, 4 und 5. A wird auf einen Ausweichplatz gesetzt. Dann werden gesetzt: D auf 1, B auf 4, E auf 2, C auf 5 und zum Schluß A auf 3. Erforderlich waren $n + 1 = 6$ Würfeltransporte, da gilt $\text{ggt}(5, 3) = 1$.

Beispiel 2: Sei $n = 6$ und $i = 2$. A kommt auf einen Ausweichplatz, C auf 1, E auf 3 und A auf 5. Nun wird B auf einen Ausweichplatz gesetzt, D kommt auf 2, F auf 4 und zum Schluß wird B auf seinen endgültigen Platz 6 gesetzt. Die Gesamtzahl der Transporte beträgt $n + 2 = 8$, da gilt $\text{ggt}(6, 2)) = 2$.

4.6 Würfelspiel

4.6.1 Aufgabenstellung

Vier Würfel mit farbigen Flächen werden aufeinandergetürmt, z. B.:



Aufgabe:
Schreibe ein Programm, welches für jeweils vier Würfel alle verschiedenen Türme ausgibt, bei denen an jeder Seitenwand keine Farbe doppelt vorkommt. Die Würfel sollen beschrieben werden in der Form:

Farbe oben
Farbe vorn, rechts, hinten, links
Farbe unten

Schicke uns Lösungen für fünf Beispiele, darunter die für folgenden Würfelturm:

gelb
grün, grün, blau, rot
gelb

blau
rot, rot, rot, gelb
grün

gelb
rot, blau, gelb, grün
blau

blau
rot, rot, blau, grün
gelb

4.6.2 Lösungsidee

Die Aufgabe ist iterativ dadurch lösbar, daß alle Konstellationen durchprobiert werden. Dazu muß jeder Würfel mit jeder Fläche viermal nach unten weisen und dabei jede der vier Seitenflächen einmal nach vorn richten. Da eine Drehung des gesamten Turms um die vertikale Achse die Turmfassaden nicht verändert, muß bei einem Würfel die Drehung um die vertikale Achse entfallen. Andernfalls wird ein korrekter Turm viermal ausgegeben, jeweils um 90 Grad gedreht.

Insgesamt müssen $(6 * 4)^3 * 6 = 82944$ Konstellationen ausprobiert werden.

Für jede gefundene Lösung gibt es natürlich noch 23 weitere, wenn man die Positionen der Würfel im Turm ändert, also etwa den obersten und den zweitobersten Würfel miteinander vertauscht. Dies war aber nicht erforderlich und wird daher im angegebenen Programm auch nicht gemacht.

Das Prüfen auf Vorliegen einer Lösung kann mit geschachtelten IFs geschehen (das ist etwas unschön), eleganter ist das folgende Verfahren:

Man ordnet den Farben die Zahlen 1, 2, 4, 8 zu und prüft, ob die Summe über jede Seitenwand des Turmes gleich $1 + 2 + 4 + 8 = 15$ ist. In beiden Fällen kann man einen Turm erst prüfen, wenn alle 4 Seiten feststehen. Verwendet man ein Array *farben: array*[*rot..blau*] *of boolean*, so kann man bereits beim Setzen des zweiten Würfels prüfen, ob der Turm gültig ist. Man setzt für jede Seite die bereits vorhandenen Farben auf *true* und prüft beim Setzen eines Würfels, ob auf einer Seite eine Farbe auftritt, die bereits *true* ist. Wenn ja, braucht man den Turm nicht weiter zu bauen.

Es kann sein, daß sich Lösungen ergeben, bei denen die Seitenwände des Turmes gleich sind, sich die Flächen oben und/oder unten aber unterscheiden. Ob dies erkannt wird, hängt davon ab, ob jede gefundene Lösung mit bereits vorher gefundenen Lösungen verglichen wird, und wie dies geschieht.

Außerdem kann es Türme geben, die absolut identisch aussehen, obwohl ein Würfel anders orientiert ist (beispielsweise wenn alle Seiten eines Würfels die gleiche Farbe haben). Im angegebenen Programm werden solche Türme als verschieden ausgegeben, um das Programm nicht zu überfrachten.

Wir verwenden den Datentyp *Wuerfel*, bei dem es sich um ein Array der Größe 6 handelt, in dem die 6 Seiten eines Würfels gespeichert werden. Dann läßt sich der Würfelturm als *turm: array*[1..4]*of Wuerfel* vereinbaren.

Die Berechnung der Türme erfolgt in der rekursiven Prozedur *Trace*(*tiefe*), in der in zwei verschachtelten Schleifen alle Kombinationen ausprobiert werden, die zu tatsächlich verschiedenen Türmen führen können.

Für das geforderte Beispiel gibt es 2 Lösungen:

```
gelb                                    gelb
gruen  gruen  blau  rot                 blau  gruen  gruen  rot
gelb                                    gelb

rot                                     rot
gelb  blau  rot  gruen                  rot  blau  gelb  gruen
rot                                     rot

gruen                                   blau
blau  rot  gelb  gelb                   gelb  rot  blau  gelb
blau                                    gruen

blau                                    rot
rot  gelb  gruen  blau                  gruen  gelb  rot  blau
rot                                     blau
```

4.6.3 Programmtext

```
program Wuerfelspiel;

uses crt;

type Farbe = (rot,gruen,gelb,blau);
     Wuerfel = array[1..6] of farbe;

var turm : array[1..4] of Wuerfel;
    ausgabe : text;
    name : string;
    loesungen : word;

function StrToFarbe(s: string; n: byte): Farbe; {wandelt String mit Farbe in}
var i,z: byte;                                  {den Typ Farbe um}
begin                                           {es wird der Teilstring vor}
  z := 1;                                       {dem n-ten Komma umgewandelt}
  if n > 1 then begin
    for i := 1 to n-1 do
```

```
      repeat inc(z) until s[z] = ',';
    inc(z)
  end;
  case upcase(s[z]) of
    'R' : StrToFarbe := rot;
    'B' : StrToFarbe := blau;
    'G' : if upcase(s[z+1]) = 'E' then StrToFarbe := gelb
            else StrToFarbe := gruen;
    else write(chr(7))
  end
end;

function FarbeToStr(farb: Farbe): string;  {wandelt Typ Farbe in String um}
begin
  case farb of
    gruen : FarbeToStr := 'grün';
    gelb : FarbeToStr := 'gelb';
    rot : FarbeToStr := 'rot';
    blau : FarbeToStr := 'blau'
  end
end;

procedure Einlesen(Name: string);      {liest Eingabedatei in Datenstruktur}
var f : text;
    i,n: byte;
    s : string;
begin
  write('Lese␣jetzt␣Würfel␣aus␣Datei␣',Name,'␣...');
  assign(f,Name);
  reset(f);
  for i := 1 to 4 do
  begin
    write('..');
    readln(f,s);
    turm[i,1] := StrToFarbe(s,1);
    readln(f,s);
    for n := 1 to 4 do
      turm[i,n+1] := StrToFarbe(s,n);
    readln(f,s);
    turm[i,6] := StrToFarbe(s,1);
    readln(f)
  end;
```

```
  writeln('..␣ok');
  close(f)
end;

procedure Ausgeben;                    {gibt Würfelturm in Textdatei aus}
var n,i: byte;
begin
  writeln(Loesungen,'.␣Lösung␣gefunden...');
  writeln(Ausgabe, Loesungen,'.␣Lösung␣gefunden:');
  for n := 1 to 4 do
  begin
    writeln(Ausgabe,FarbeToStr(turm[n,1]));
    for i:=1 to 4 do
    begin
      write(Ausgabe,FarbeToStr(turm[n,i+1]));
      if i < 4 then write(Ausgabe,',');
    end;
    writeln(Ausgabe);
    writeln(Ausgabe,FarbeToStr(turm[n,6]));
    writeln(Ausgabe)
  end
end;

procedure DreheWuerfelVert(w: byte);
{dreht den w-ten Würfel des Turms vertikal (nach hinten)}
var zwi: Farbe;
begin
  zwi := turm[w,1];
  turm[w,1] := turm[w,5];
  turm[w,5] := turm[w,6];
  turm[w,6] := turm[w,3];
  turm[w,3] := zwi
end;

procedure DreheWuerfelHoriz(w: byte);
{dreht den w-ten Würfel des Turms horizontal (nach rechts)}
var zwi: Farbe;
begin
  zwi := turm[w,2];
  turm[w,2] := turm[w,5];
  turm[w,5] := turm[w,4];
  turm[w,4] := turm[w,3];
```

```
  turm[w,3] := zwi
end;

function TurmGueltig(tiefe: byte): boolean;        {überprüft, ob der Turm bis}
var s,w : byte;                                    {zum Würfel Tiefe gemäß der}
    zwi : boolean;                                 {Aufgabenstellung gültig ist}
    farben: array[rot..blau] of boolean;
begin
  TurmGueltig := true;
  for s := 2 to 5 do
  begin
    FillChar(farben, sizeof(farben), false);
    for w := 1 to tiefe do
    begin
      if farben[turm[w,s]] = true then
        TurmGueltig := false;
      farben[turm[w,s]] := true
    end
  end
end;

function KeineSymmetrie(tiefe, seite: byte): boolean;
begin
  if (tiefe > 1) or ((tiefe = 1) and (seite = 1)) then
    KeineSymmetrie := true
  else
    KeineSymmetrie := false
end;

procedure Trace(tiefe: byte);
var s,h,x,y: byte;                                 {rekursives Backtracking}
begin
  if tiefe = 5 then begin
    inc (Loesungen);
    Ausgeben
  end
  else begin
    for h := 1 to 6 do
    begin
      for s := 1 to 4 do                           {4 x horizontal drehen}
      begin
        if (TurmGueltig(tiefe) and KeineSymmetrie(tiefe,s)) then
```

```
            Trace(tiefe + 1);                                    {rekursiv}
          DreheWuerfelHoriz(tiefe)
        end;
        case h of
          1..3: DreheWuerfelVert(tiefe);
          4 : begin
                  DreheWuerfelHoriz(tiefe);
                  DreheWuerfelVert(tiefe)
                end;
          5 : begin
                  DreheWuerfelVert(tiefe);
                  DreheWuerfelVert(tiefe)
                end;
        end  {case}
      end  {for h := 1 to 6}
    end  {else}
  end;  {Trace}

begin
Loesungen := 0;
if ParamCount <> 1 then              {Kommandozeilenparameter auswerten}
  Name := 'wuerfel'
else
  Name := ParamStr(1);
Einlesen(Name+'.in');
assign(Ausgabe,Name+'.out');                          {Ausgabedatei öffnen}
writeln('Ausgabe␣in␣Datei␣'+Name+'.out');
rewrite(Ausgabe);
Trace(1);                                                        {Rekursion}
close(Ausgabe);
writeln('Fertig!!');
readln
end.
```

4.7 Kuriositäten und besonders nette Formulierungen

(Rechtschreibfehler gehen auf das Konto der Einsender.)

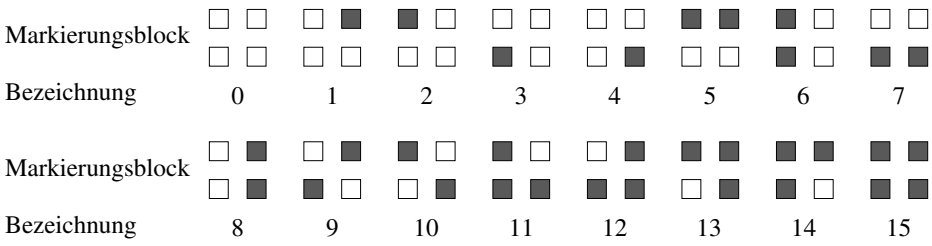
- Wenn der Rechner was passendes gefunden hat, entkommt er der While-Schleife.
- Bei unserer Betrachtung haben wir psycho/soziologische Faktoren außer acht gelassen.
- vercodet — (*das tut wirklich weh!*)
- Exklusiv-Verodert werden — (*das ist auch nicht viel besser*)
- Kein Schiff kann sich überlappen.
- Einen echten Durchschnittswert kann es aber nicht geben, da es immer auf den Spieler ankommt (und dessen parapsychologische Fähigkeiten).
- $\mathrm{ggt}(n, i) \cdot 2 + n - \mathrm{ggt}(n, i)$
- Aber dies wäre wohl etwas kopflos, wobei mir Maria Stuart sicher recht geben würde.
- *Einer der Teilnehmer schickte noch ein Blatt hinterher und vermerkte dazu, seine eigentliche Einsendung befinde sich in einem braunen DIN A4-Umschlag. Davon hatten wir ca. 800 Stück ...*

5 13. Wettbewerb: Die Aufgaben der 2. Runde

5.1 Promesiphus

5.1.1 Aufgabenstellung

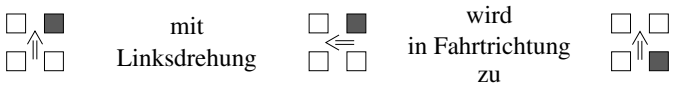
Die Stadtväter von Siphus haben von dem berühmten Verkehrsleitsystem Prometheus gehört und wollen etwas Entsprechendes in ihrer Stadt einführen. Weil das aber eine wahre Sisiphusarbeit ist, nennen sie ihr System Promesiphus. Zur Verkehrssituation in dieser Stadt: Jede Straße besteht aus zwei Fahrbahnen, eine für jede Richtung (es gibt also keine Einbahnstraßen). Auf den Fahrbahnen sind in regelmäßigen Abständen Fahrbahnmarkierungen verteilt. Die mittleren Markierungen gelten sowohl für die rechte wie auch für die linke Fahrbahn. Aus je 4 im Quadrat angeordneten Fahrbahnmarkierungen, im folgenden Markierungsblöcke genannt, wird nach vorgegebenen Kodierungen die auszuführende Bewegung abgelesen. Mögliche Markierungsblöcke sind:



Die folgenden vier Bewegungen sollen ausgeführt werden können:

G	einen ganzen Markierungsblock nach vorne
H	einen halben Markierungsblock nach vorne
L	eine Drehung um 90° nach links
R	eine Drehung um 90° nach rechts

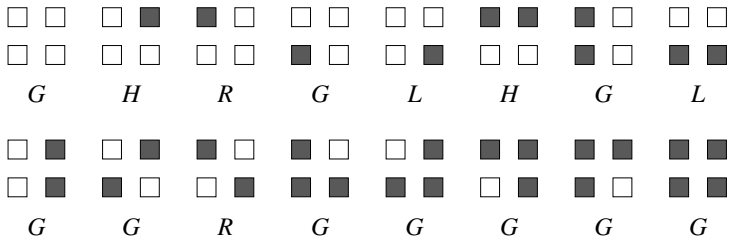
Die Drehungen erfolgen innerhalb der Markierungsblöcke, d. h. es wird keine Vorwärts- oder Rückwärtsbewegung ausgeführt.
Wenn beispielsweise der Markierungsblock 1 eine Linksdrehung ist, wird als nächstes der Markierungsblock 4 gelesen.



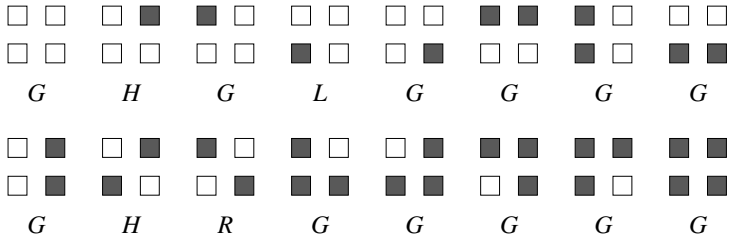
Es darf auf eine Drehung nicht wieder eine Drehung folgen. Zwei Links- bzw. Rechtsdrehungen lassen ein Fahrzeug auf der gleichen Spur wenden. Das ist verboten, da es dann gegen die Fahrtrichtung steht. Die Kombination einer Links- mit einer Rechtsdrehung führt zu einer endlosen Schleife.

Es dürfen auch nicht zwei halbe Schritte aufeinander folgen.

Die Stadtväter haben ein System entwickeln lassen. Die Abbildungen 5.1 und 5.2 zeigen einen vereinfachten Stadtplan mit Markierungen und Verkehrszeichen, die Fahrtrichtungen vorschreiben. Der im ersten Plan eingezeichnete Weg von **Norden nach Westen** ergibt sich, wenn den 15 Markierungsblöcken die vier Bewegungen wie folgt zugeordnet werden:



Entsprechend ergibt sich der im zweiten Plan eingezeichnete Weg (gleicher Stadtplan, nur anderer Weg) von **Norden nach Osten** durch folgende Zuordnung:



Aufgaben:

1. Programmieren Sie ein System, das den angegebenen Weg nach Eingabe des Plans auf Korrektheit und die Zuordnung von Bewegungen zu Markierungsblöcken auf Widerspruchsfreiheit überprüft.

2. Bestimmen Sie einen Weg von Norden nach Süden und überprüfen Sie diesen Weg mit Hilfe Ihres Systems.

3. Man kann nicht jedem Markierungsblock eine beliebige Bewegung zuordnen, da es bestimmte Kombinationen gibt, die sich ausschließen.

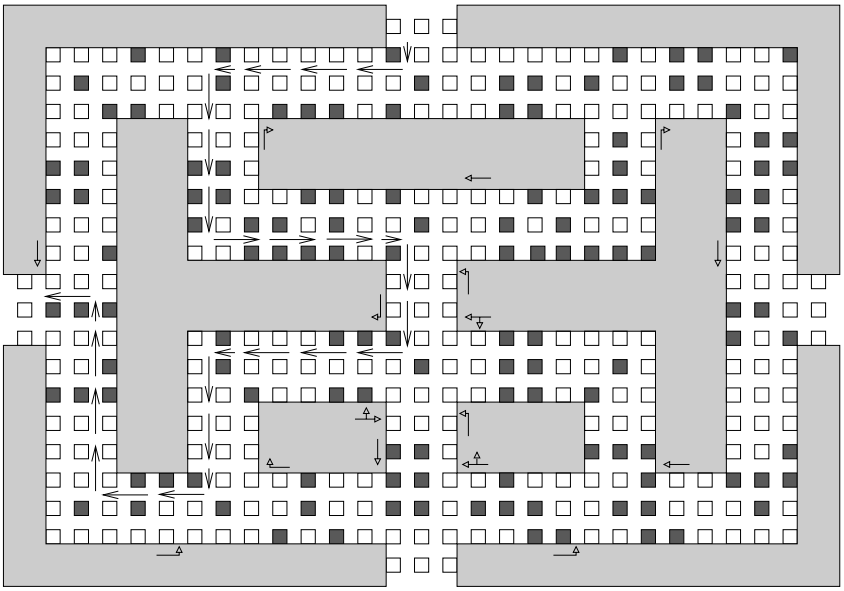


Abbildung 5.1: Der Weg von Norden nach Westen

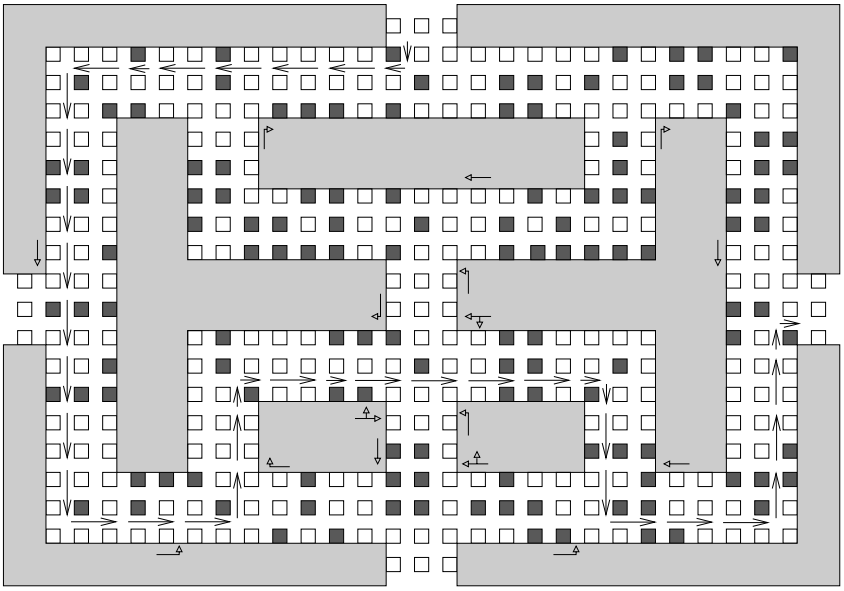


Abbildung 5.2: Der Weg von Norden nach Osten

Beispiel: Der Markierungsblock 0 (s. Abb.) kann keine Drehung sein, weil er sich auch nach der Ausführung beliebig vieler Drehungen nicht verändert.

Geben Sie alle widerspruchsfreien und vollständigen Zuordnungen von Bewegungen zu den Markierungsblöcken an. Eine Zuordnung ist dabei genau dann vollständig, wenn

- jedem Markierungsblock genau eine Bewegung zugeordnet ist, und wenn
- jeder Bewegung mindestens ein Markierungsblock zugeordnet ist.

4. Skizzieren Sie ein mögliches Prinzip, nach dem man sich überlappende Wegemarkierungen anlegen kann.

5.1.2 Lösungsidee

Allgemeines:

Das Programm muß die Verkehrszeichen berücksichtigen (sonst wären sie im Stadtplan nicht angegeben).

Wenden auf der Fahrbahn (also Linksdrehung, Halbschritt, Linksdrehung) ist nicht explizit verboten und daher prinzipiell erlaubt. Es muß aber zumindest angegeben werden, daß solche Drehungen erlaubt sind. Bei Kreuzungen muß man dann aber aufpassen, daß niemand zum Geisterfahrer wird.

Der vorgegebene Stadtplan kann fest im Programm stehen; ein Stadtplaneditor kann als Erweiterung programmiert werden.

Zu Teilaufgabe 1:

Zur Wegeüberprüfung gibt es verschiedene Vorgehensweisen (wobei wir annehmen, daß der Stadtplan vorliegt):

- Es werden eine Zuordnung und ein Startpunkt eingegeben. Das Programm gibt dann aus: "Weg endet im Norden / Süden / Westen / Osten / an einer Wand."
  Oder es wird zusätzlich noch das Ziel eingegeben und das Programm meldet "geht" bzw. "geht nicht".

- Zusätzlich zur Zuordnung und dem Startpunkt wird der Weg eingegeben (das ist etwas überspezifiziert und daher nicht so schön), und das Programm meldet Übereinstimmung.

- Oder man gibt einen Weg ein und das Programm liefert die sich daraus ergebende Zuordnung, indem es jedem durchfahrenen Markierungsblock die Bewegung zuordnet, die der Weg erfordert. Eine so erzeugte Zuordnung muß natürlich noch auf Widerspruchsfreiheit geprüft werden.

Bei der Prüfung einer Zuordnung auf Widerspruchsfreiheit sind folgende Punkte zu beachten:

1. Die Blöcke 0 und 15 dürfen keine Drehungen sein.
2. Wenn Block 0 ein Halbschritt ist, dürfen die Blöcke 1, 2 und 5 keine Halbschritte sein.
3. Entsprechendes gilt für Block 15 und die Blöcke 7, 11 und 12.
4. Entsprechende Bedingungen, welche Blöcke nicht gleichzeitig Halbschritte sein dürfen, können für jeden Block aufgestellt werden.
5. Wenn Block 1 eine Rechtsdrehung ist, darf Block 2 keine Drehung sein.
6. Wenn Block 1 eine Linksdrehung ist, darf Block 4 keine Drehung sein.
7. Entsprechendes gilt für Block 2 und die Blöcke 3 bzw. 1.
8. Entsprechendes gilt für Block 3 und die Blöcke 4 bzw. 2.
9. Entsprechendes gilt für Block 4 und die Blöcke 1 bzw. 3.
10. Die Bedingungen 6 – 9 für sich ausschließende Kombinationen für die Blöcke 1, 2, 3 und 4 gelten entsprechend für die Blöcke 5, 6, 7 und 8 und für die Blöcke 11, 12, 13 und 14.
11. Wenn Block 9 eine Drehung ist, darf Block 10 keine Drehung sein und umgekehrt.
12. Es darf keine Kombination von Zuordnungen geben, so daß Linksdrehung – Halbschritt – Rechtsdrehung möglich ist, da dies zu Geisterfahrern führt.
13. Ebenso darf es nicht geben Rechtsdrehung – Halbschritt – Rechtsdrehung, da man damit gegen eine Wand fährt.

Man kann sogar noch weiter gehen und eine gegebene, widerspruchsfreie Zuordnung am Stadtplan prüfen, ob sie Drehungen erlaubt, die ein Fahrzeug unmittelbar vor einer Wand enden lassen (das dann nur noch auf diese Wand zufahren kann) oder ob sie Geradeaus-Schritte erlaubt, wenn man bereits vor einer Mauer steht.

Die beiden vorgegebenen Wege (von Norden nach Westen und von Norden nach Osten) sind korrekt, d. h. sie entsprechen den angegebenen Zuordnungen von Bewegungen zu Markierungsblöcke und diese Zuordnungen sind widerspruchsfrei.

Zu Teilaufgabe 2:

Im angegebenen Plan gibt es keine korrekte Zuordnung, die einen Weg von Norden nach Süden ermöglicht. Wegen der Verkehrszeichen kann man nur von Norden oder von Westen her auf den Südausgang zufahren. Von Osten kommend verbietet ein Verkehrszeichen das Linksabbiegen. Aus dem Norden kann man aber auch nicht kommen, da man am linken T-förmigen Gebilde rechts abbiegen muß; zwischen dem linken T und dem kleinen Rechteck unten links von Westen kommend darf man nicht nach rechts, also in Richtung Süden, abbiegen. Es bleibt nur ein Weg von Norden nach Süden übrig, siehe Abbildung 5.3.

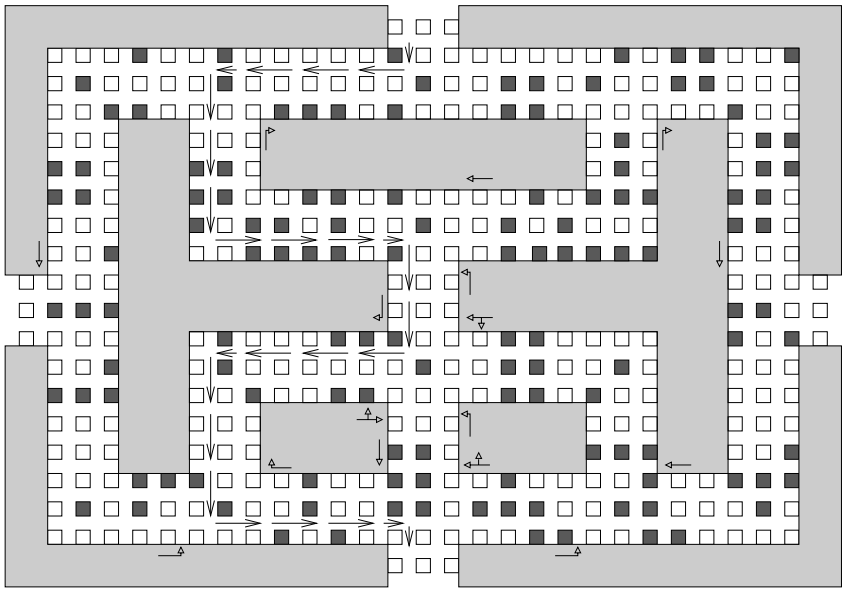


Abbildung 5.3: Ein Weg von Norden nach Süden (ohne Überprüfung der Zuordnung)

Für diesen Weg gibt es jedoch keine korrekte Zuordnung von Bewegungen zu Markierungsblöcken. An der nördlichen Einfahrt wird Block 1 ein Halbschritt zugeordnet, Block 10 eine Rechtsdrehung. Das ist notwendig, denn Linksabbiegen führt nicht zum gewünschten Ziel im Süden. Am unteren Ende des linken T landet man entweder in Block 1, dann muß ihm ein ganzer Schritt zugeordnet werden,

oder man landet in Block 10, dem dann ein Halbschritt zugeordnet werden muß. Beides steht im Widerspruch zur bereits erfolgten Zuordnung.

Streicht man das Verkehrszeichen am unteren Ende des rechten T, so kann man an dieser Ecke rechts abbiegen und erhält den in Abbildung 5.4 gezeigten Weg.

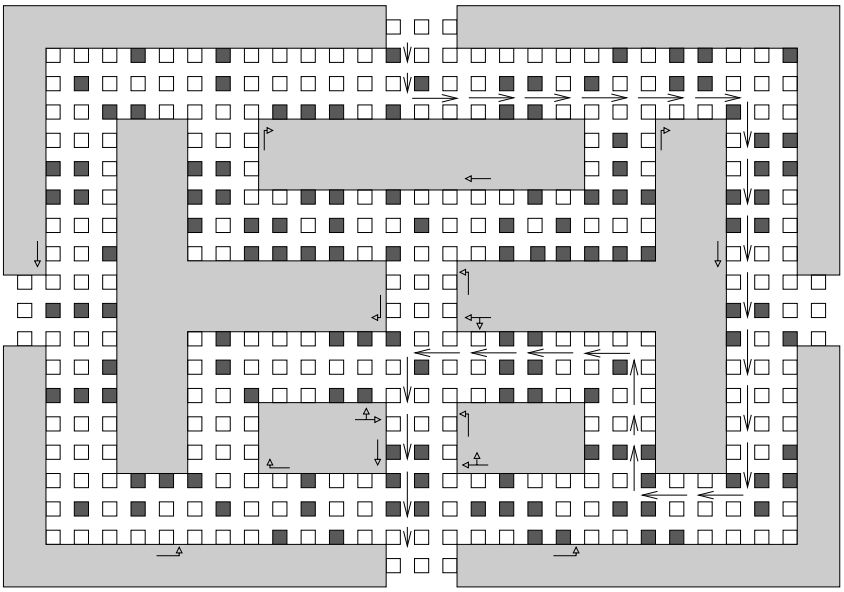
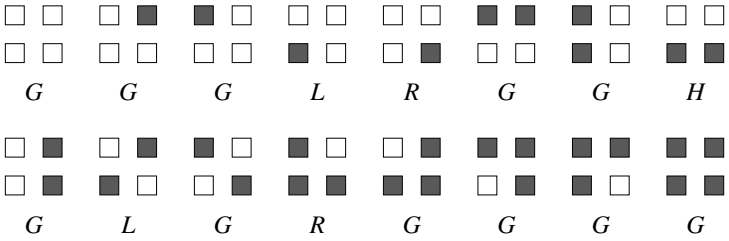


Abbildung 5.4: Ein Weg von Norden nach Süden nach Entfernen eines Verkehrszeichens

Für diesen Weg ist die folgende Zuordnung gültig:



Zu Teilaufgabe 3:

Diesen Aufgabenteil kann man völlig unabhängig vom gegebenen Stadtplan lösen. Beispielsweise kann man ein Programm verwenden, das rekursiv alle möglichen

Zuordnungen von Bewegungen zu Markierungsblöcken erzeugt und jeweils prüft (wie in Teilaufgabe 1 angegeben), ob die Zuordnung widerspruchsfrei ist. Dabei müssen nicht alle theoretisch möglichen 4^{16} Zuordnungen vollständig erzeugt werden, sehr häufig kann man frühzeitig abbrechen, weil sich bereits ein Widerspruch ergeben hat bzw. manche Zuordnungen von vornherein nicht erlaubt sind. Und man kann nach Zuordnung einer Bewegung zu einem Markierungsblock die möglichen Bewegungen für weitere Blöcke bereits einschränken; siehe die zu Teil 1 auf Seite 107 aufgeführten Bedingungen.

Die erzeugten widerspruchsfreien Zuordnungen müssen noch auf Vollständigkeit geprüft werden. Das läßt sich erreichen, indem das Programm immer jedem Block eine Bewegung zuordnet, also alle 16 Blöcke bearbeitet. Anschließend wird dann geprüft, ob tatsächlich jede Bewegung berücksichtigt wurde. Das Verfahren kann man etwas beschleunigen, wenn bereits bei den Zuordnungen von Bewegungen zu den letzten vier Blöcken geprüft wird, ob alle Bewegungsmöglichkeiten verwendet werden können.

Insgesamt erhält man 1.224.758 gültige, also vollständige und widerspruchsfreie, Zuordnungen.

Man kann den Aufgabenteil 3 auch auf den gegebenen Stadtplan beziehen und alle Zuordnungen ermitteln, die im Stadtplan sinnvolle Wege ergeben, also Wege, die an einem der Eingänge beginnen und an einem anderen enden.

Läßt man wiederum den Pfeil im unteren Ende des rechten T weg, ergeben sich mehrere sinnvolle Zuordnungen.

Dabei unterscheiden sich Wege mit gleichem Start- und Zielpunkt nicht in der Streckenführung, sondern nur in der Schrittfolge, d. h. bei einem Weg wird etwa ein Halbschritt gemacht, gefolgt von einem ganzen Schritt, während bei einem anderen Weg an der entsprechenden Stelle zuerst der ganze Schritt kommt, dann der halbe.

Wir geben jeweils vollständige Zuordnungen an, erwähnen aber auch, welche Blöcke nicht durchfahren werden, für die also — natürlich unter Berücksichtigung der den anderen Blöcken zugeordneten Bewegungen — mehrere Bewegungen möglich sind.

Weg	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	nicht bef.
N → W	G	H	R	G	L	H	G	L	G	G	R	G	G	G	G	G	11,14
N → S	G	G	G	L	R	G	G	H	G	L	G	R	G	G	G	G	5,8,13,14
N → O	G	H	G	L	G	G	G	G	G	H	R	G	G	G	G	G	
	G	H	G	L	G	H	G	G	G	H	R	G	G	G	G	G	6,11,13,14
	G	H	G	L	G	H	G	G	G	H	R	G	G	G	G	H	
O → S	G	G	G	L	H	G	L	H	G	L	G	R	G	G	G	G	1,8,13,14
	G	G	G	L	G	H	L	H	G	L	G	R	G	G	G	G	1,4,8,13,14

Von Westen und von Süden aus gibt es keine Wege.

Zu Teilaufgabe 4:

Gefragt ist, wie man einen beliebigen Stadtplan markieren (d. h. schwarze Kästchen verteilen) kann, damit sich durch unterschiedliche Zuordnungen von Bewegungen zu Blöcken möglichst viele unterschiedliche, aber vernünftige Wege ergeben. Verkehrszeichen müssen nicht berücksichtigt werden. Dabei geht es nicht um einen vollständigen Algorithmus, sondern nur darum, einige Ideen vorzustellen.[7]

Das Ziel ist nicht, mit *einer* Zuordnung mehrere Wege fahren zu können, denn wenn man das erreichen will, kann den Markierungsblöcken keine eindeutige Bewegung mehr zugeordnet werden. Zur Vermeidung eines Verkehrschaos ist dies aber notwendig.

Man beginnt mit einem quasi leeren Plan, d. h. mit einem Plan, der vollständig mit dem Block 0 besetzt ist. Es ist zweckmäßig, diesem Block die Bedeutung "ganzer Schritt" zuzuordnen. Es gibt verschiedene Gruppen von Markierungsblöcken, etwa 1 – 4, 5 – 8, 9/10 und 11 – 14, und Paare, etwa 4 und 13, die sich besonders gut als Kurven eignen. An die Eckpunkten der Routen, die man im Plan realisieren will, setzt man die verschiedenen Kurvenblöcke, wobei man immer die gleichen Blöcke verwendet. Man versucht also, mit einem R- und einem L-Block auszukommen.

Dann besetzt man die Kreuzungen so, daß sinnvolle Bewegungen für jede Richtung möglich sind, aus der man an die Kreuzung herankommt.

Können Wege an Ecken oder Kreuzungen nicht genau auf dem gewünschten Markierungsblock ankommen, sondern treffen auf den direkt davorliegenden Block, muß diesem die Bewegung Halbschritt zugeordnet werden.

[7] Wir folgen dabei teilweise der Darstellung von Utz Bacher.

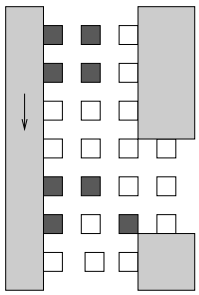
Datenstrukturen

Nach den Erläuterungen zu Wegen und korrekten Zuordnungen wollen wir nun noch einige Hinweise auf mögliche Datenstrukturen geben.

Jeder Markierungsblock besteht aus vier Punkten, die jeweils zwei Zustände (schwarz oder weiß) besitzen können. Man kann einen Block daher als vierstellige Binärzahl auffassen; schwarze Punkte seien durch 1, weiße durch 0 dargestellt. Betrachtet man den Punkt rechts oben als niedrigstwertiges Bit und geht die übrigen Punkte im Uhrzeigersinn durch, erhält man beispielsweise für den Markierungsblock 1 der Aufgabenstellung ebenfalls die Nummer 1, für den Block 2 die Nummer 8 und für den Block 12 die Nummer 7. Nun speichert man die aktuelle Zuordnung von Bewegungen zu Markierungsblöcken in einem Array der Größe 16, das mit den neuen Nummern indiziert wird (beginnend bei 0).

Man erhält also eine andere Numerierung als in der Aufgabenstellung, kann aber dafür beim Durchfahren des Stadtplans sehr schnell feststellen, welche Bewegung dem Block zugeordnet ist, in dem man sich gerade befindet. Dazu interpretiert man die Punkte wie beschrieben und erhält dann mit einem Zugriff auf das Array die zugehörige Bewegung.

Die beschriebene Umnumerierung der Markierungsblöcke ist sinnvoll, wenn man den Stadtplan in einzelne Punkte unterteilt und nicht in Markierungsblöcke. Man verwendet für den Stadtplan ein zweidimensionales Array. Ein Zelleneintrag kann die Werte 0 bzw. 1 für einen weißen bzw. für einen schwarzen Markierungspunkt, 2 für eine Mauer oder eine der Zahlen 3 – 8 für eines der sechs möglichen Verkehrszeichen enthalten. Die Verkehrszeichen kann man jeweils so angeben, wie sie in Fahrtrichtung auftreten, da sie nur dann von Bedeutung sind. Einen kleinen Ausschnitt aus dem vorgegebenen Stadtplan in dieser Darstellung zeigt Abbildung 5.5, wobei dem Verkehrszeichen noch keine Nummer zugeordnet wurde.



2	1	1	0	2
2	1	1	0	2
V	0	0	0	2
2	0	0	0	0
2	1	0	0	0
2	1	0	1	0
2	0	0	0	2

Abbildung 5.5: Darstellung des Stadtplans aus einzelnen Punkten

Zu einem Fahrzeug, das durch die Stadt fährt, benötigt man jeweils seine aktuelle Position und Fahrtrichtung. Zweckmäßigerweise positioniert man das Fahrzeug auf einem der vier Punkte des Markierungsblocks, in dem es sich gerade befindet, beispielsweise auf dem vorderen rechten. Für ein Fahrzeug an Position (x, y) mit Fahrtrichtung Norden (und Koordinatenursprung unten links) müssen dann die Zellen (x, y), $(x, y\text{-}1)$, $(x\text{-}1, y\text{-}1)$ und $(x\text{-}1, y)$ gelesen werden, um den aktuellen Markierungsblock zu bestimmen. Außerdem müssen noch Zellen rechts inspiziert werden, da dort möglicherweise Verkehrszeichen stehen. Dabei genügt es nicht, nur jeweils die Zelle $(x{+}1, y)$ zu betrachten, also die Zelle rechts neben dem angenommenen Fahrzeug, denn mit einem ganzen Schritt könnten dann Verkehrszeichen überfahren werden. Also muß eine weitere Zelle inspiziert werden, wobei es sinnvoll ist vorauszuschauen, also die Zelle $(x{+}1, y{+}1)$ zu betrachten.

Statt Verkehrszeichen explizit als solche zu speichern, kann man auch zu jedem Markierungsblock (also in jeder Zelle) speichern, in welche Richtungen er verlassen werden darf. Damit entfällt auch die Prüfung, ob man gegen eine Wand gefahren ist, und ein Fahrzeug kann dann auch nicht zum Geisterfahrzeug werden.

Verkehrszeichen wurden hier immer direkt vor der Kreuzung plaziert, wie es in der Straßenverkehrsordnung vorgeschrieben ist. Will man ermöglichen, daß sie auch etwas weiter entfernt stehen können, so muß sich ein vorbeifahrendes Fahrzeug das Verkehrszeichen bis zur nächsten Kreuzung merken und dann entsprechend weiterfahren.

Übrigens bietet es sich an, beim Abfahren eines Weges die bereits durchfahrenen Blöcke zu markieren, so daß man feststellen kann, wenn man im Kreis fährt, also einen Block zum zweiten Mal in der gleichen Richtung durchfährt.

5.2 Testreiheneditor

5.2.1 Aufgabenstellung

Die aus einem Versuch gewonnenen Datenreihen sollen per Computer ausgewertet werden. Dazu müssen sie zunächst einmal manuell eingegeben, überprüft und korrigiert werden. Die Werte bestehen aus endlichen Dezimalzahlen mit unterschiedlicher Stellenzahl. Sie sind zu Zeilen zusammengefaßt und die Zeilen wieder zu Seiten. Das Erfassungs- und Korrekturprogramm soll sehr flexibel sein. Die Zeichenzahl zur Darstellung der Zahlen, die Anzahl der Zahlen in einer Zeile und die Anzahl der Zeilen auf einer Seite stehen nicht von vornherein fest.

Die erfaßten Testseiten sollen in einer Datei gespeichert werden. Natürlich sollen diese auch mit demselben Programm wieder geladen werden können, z. B. zum Korrigieren.

Aufgabe:

Beschreiben Sie mindestens zwei Strategien zur Realisierung des geforderten Editors.

- Welche Probleme gibt es bei der Realisierung zu lösen?
- Welche Probleme ergeben sich aus unterschiedlichen Zahlendarstellungen?
- Welche Datenstrukturen kommen in Frage?
- Warum kommen gerade diese in Frage?
- Welche Datenstruktur wählen Sie, wenn möglichst wenig Speicherplatz benutzt werden soll?
- Welche Dateistruktur wählen Sie?

Das Erstellen eines Programms bringt keine zusätzlichen Punkte!

5.2.2 Lösungsidee

Unterschiedliche Strategien können beispielsweise die Minimierung des Speicherplatzes oder die Minimierung der Laufzeit bei Änderungen oder möglichst große Benutzerfreundlichkeit sein.

Wir stellen im folgenden einige Strategien vor, die jeweils auf eine Realisierung in Turbo Pascal abzielen.

Erste Strategie:[8]

Der Benutzer muß vor dem Bearbeiten der Zahlen zwischen "1. Zahleingabe", "2. Zahl korrigieren" (dazu zählt auch das Überprüfen einer Zahl) und "3. Formatierung der Zahlen" wählen.

1. Die Zahlen werden, so wie sie aus dem Versuch gewonnen wurden, eine nach der anderen eingegeben, wobei für die Eingabe nur Ziffern, das Komma und das Minuszeichen (eventuell bei einer anderen Zahlendarstellung noch das Zeichen "E") zugelassen sind und jede Zahleingabe mit einer vereinbarten Taste (z. B. "Return") abgeschlossen wird. Nach dieser Taste wird die eingegebene Zahl im Speicher abgelegt, d. h. an die Zahlenliste im Speicher angefügt. Ist eine Zahl eingegeben, kann sie im Eingabemodus nicht mehr verändert werden. Erst wenn alle Zahlen eingegeben sind, kann (durch eine vereinbarte Tastenkombination) der Eingabemodus verlassen werden.

2. Um Zahlen zu korrigieren, müssen schon Zahlen eingegeben worden sein. Mit den Cursor-Tasten kann man dann von Anfang bis Ende der Zahlenliste (im Speicher) wandern und die Zahlen überprüfen und gegebenenfalls verändern. Aufgrund der Datenstruktur sind Funktionen wie Löschen, Einfügen von Zahlen oder beliebige Veränderung der Stellenzahl einer Zahl möglich.

3. Erst wenn alle Zahlen des Versuchs eingegeben worden sind, ist die Formatierung sinnvoll. Dabei werden alle Zahlen (die bisher nur im Speicher abgelegt sind) anhand von eingebbaren Optionen wie Zahlen/Zeile oder Zeilen/Seite am Monitor dargestellt.

Die erfaßten Daten werden der Einfachheit halber bei Beenden des Editors in eine Datei gespeichert und beim Neustart des Editors automatisch aus dieser Datei in den Speicher geladen. Natürlich können Laden und Speichern auch Menüpunkte innerhalb des Editors sein.

Probleme bei der Realisierung:

Formatierung: Da die Zahlen in einer Liste stehen und ohne Formatierungsanweisung im Hauptspeicher abgespeichert werden, muß bei jeder Testseiten-Vorschau die Formatierung durchgeführt werden, was sicherlich viel Zeit in Anspruch nimmt. Ohne Formatierung steht immer nur eine Zahl aus der Liste für eine Bearbeitung zur Verfügung. Außerdem ist bei der Dateistruktur für Formatierungsangaben kein Platz vorgesehen, so daß eine zweite Datei mit Optionen wie Zeilen/Seite oder Zahlen/Zeile sinnvoll

[8] nach einem Vorschlag von Matthias Köppen

ist, da sonst alle Optionen neu angegeben werden müßten.
Korrektur: Da es unter Umständen lange dauert, bis man von Anfang bis Ende einer Liste wandert, ist eine Funktion, die zu einer bestimmten Zahl in der Liste springt, notwendig. Mit einem Zähler könnte man dann z. B. zur 100. Zahl springen, wenn man die Zahlen davor schon überprüft hat. Ein weiteres Problem ist das Löschen oder Einfügen von Zahlen. In einem Array z. B. ist das nur Verschieben vieler Felder möglich, was sehr lange dauert.

Zahlendarstellung: Vor der Zahleneingabe muß man definieren, welches Zahlenformat benutzt werden soll. Man kann z. B. festlegen, daß als Standard Dezimalzahlen mit oder ohne Dezimalpunkt eingegeben werden dürfen. Natürlich wären theoretisch auch andere Zahlensysteme möglich. Für die Abspeicherung muß diese Folge von Ziffern (d. h. Folge von Char, wobei die zehn Ziffern, Komma und Minuszeichen als Char aufgefaßt werden) in Fließkommaschreibweise umgewandelt werden. Jede Zahl besteht also intern aus einem Exponenten zur Basis 10 und einer ganzen Zahl (Mantisse).

Abspeicherung der Zahlen: Die einfachste Möglichkeit wäre das Festlegen eines Bereiches und einer Maximalgenauigkeit der Zahlen. So wäre z. B. eine Reservierung von sechs Byte (= 48 Bit) je Zahl möglich, was dem Real-Typ entspricht. Diese Zahlenabspeicherung hat den Nachteil, daß ein Bereich vorgeschrieben ist und ein Genauigkeitsverlust auftritt.

Noch einfacher wäre das Abspeichern der eingegebenen Zahl als String bzw. als Folge von Char, was zu keinem Genauigkeitsverlust führen würde, aber sehr viel Speicherplatz in Anspruch nimmt. Die Zahl 12345 z. B. nimmt als Integer nur zwei Byte ein, als Folge von Char jedoch fünf Byte.

Besser ist folgende Lösung: Wie bereits erläutert, setzt sich jede Zahl aus einer ganzen Zahl, der Mantisse, und einem Exponenten zur Basis 10 zusammen. Eine Zahl kann dargestellt werden als ein Zeiger auf folgenden Record:

```
type dynamisch = ^string;
     zahl      = ^zahltyp;
     zahltyp   = record
                    exponent : integer;
                    mantisse : dynamisch
                 end;
var eingabe    : zahl;
```

Der Exponent wird in `eingabe^.exponent` gespeichert, so daß hier ein sehr großer Bereich vorgegeben ist. Die ganze Zahl (Mantisse) wird in einen

String umgewandelt, der jedoch eine dynamische Länge hat. Dieser String ist auf 255 Stellen begrenzt.

Datenstruktur: Um eine Liste von Zahlen abzuspeichern, kann man beispielsweise ein `Array[1..n] of zahl` anlegen, wobei dann jedoch nur n Zahlen eingeben werden können. Günstiger ist auch hier eine verkettete Liste, in die problemlos Zahlen eingegeben, gelöscht oder eingefügt werden können.

Durch verkettete Listen sowohl bei der Zahlenabspeicherung als auch bei der Zahlenliste wird nur soviel Speicherplatz deklariert, wie von den eingegebenen Zahlen auch benötigt wird, plus dem Speicherplatz für die Zeigervariablen. Andere Datenstrukturen (wie Arrays oder dynamische Strings) belegen unnötig viel Speicherplatz und haben den Nachteil, daß der Zahleneingabe in Stellenzahl oder Genauigkeit Grenzen gesetzt sind.

Dateistruktur: Da verkettete Listen nicht in eine Datei geschrieben werden können (d. h. `file of zahl` ist zwar möglich, speichert aber nur die Adressen der Zahlen und nicht die Zahlen an sich), muß die Datei untypisiert sein, denn schließlich sind auch die Zahlen unterschiedlich lang. Um Platz zu sparen und nicht für jede Zahl die gleiche Anzahl von Byte festzulegen, müssen die Byte, die eine Zahl für die Mantisse benötigt, in der verketteten Liste gezählt werden und als Längenbyte zu jeder Zahl abgespeichert werden. Dies könnte man sich ersparen, wenn ein vereinbartes Sonderzeichen als Ende einer Mantisse einer Zahl vereinbart wurde. Doch ist ein vereinbartes Sonderzeichen als Trennungszeichen in diesem Fall unmöglich, da das vereinbarte Sonderzeichen selbst als Byte in einer Mantisse vorkommen könnte, was zu Fehlern führt.

Zweite Strategie:[9]

Der Programmieraufwand wird gering gehalten. Zahlen werden nicht auf ihre Gültigkeit überprüft. Es werden lediglich Zeichen von der Tastatur entgegengenommen und in die Datenstruktur eingetragen. Daher können die Zahlendarstellungen also keine Probleme ergeben. Zur Darstellung der Zeilen- und Seitenstruktur werden drei Trennzeichen verwendet: „neue Zahl“ („ “), „Neue Zeile“ („$“) und „neue Seite“ („_“). Die Trennzeichen werden durch Tippen der jeweiligen Taste geschrieben. Die Eingabe eines jeden Zeichens muß also auf Trennzeichen geprüft werden. „Neue Zeile“ soll ebenso durch Eingabe von <Enter> bewirkt werden können.

Der gesamte Bildschirm wird ausgenutzt. Eine Bildschirmzeile entspricht nicht notwendigerweise einer Zahlenzeile. Wenn eine Zahlenzeile länger ist als eine Bild-

[9] von Lothar Oppor

schirmzeile, wird stattdessen einfach auf der nächsten Bildschirmzeile weiter geschrieben. Am Ende einer Zahlenzeile wird das Zeichen „neue Zeile“ auf den Bildschirm geschrieben, auch wenn es als erstes Zeichen einer Bildschirmzeile erscheint.

Es gibt ein Funktionsmenü mit folgenden Funktionen: <Neue Datei anlegen>, <Datei öffnen>, <Datei speichern>, <Bearbeitung beenden>, <Einfügemodus einschalten>, <Einfügemodus ausschalten>. Der Cursor kann mit den üblichen Tasten gesteuert werden: <Nach rechts>, <Nach links>, <Nach unten> und <Nach oben>. Diese Steuertasten beziehen sich auf Bildschirmzeilen. Eine weitere Tastenfunktion <Bild nach unten> wirkt auf die Zahlenzeile. Seitenweises Blättern wird als <SHIFT><Bild nach unten> realisiert.

Als Dateistruktur dient `File of Character`. Es gibt, außer beim Erstellen einer neuen Datei, immer eine Quelldatei, aus der die zu editierenden Zahlen stammen, und eine Zieldatei, in die die editierten Zahlen geschrieben werden. Blättern ist nur nach vorn erlaubt. Dadurch bereiten Einfügungen oder Löschungen keine Schwierigkeiten.

Als Datenstruktur wird ein `Array of Character` verwendet, welches so viele Zeichen aufnehmen kann, wie auf den Bildschirm passen. Dieses Array dient als Editierpuffer. Die Inhalte von Bildschirm und Editierpuffer sind gleich. Wenn eine Zahlenzeile nicht mehr auf den Bildschirm paßt, wird zahlenzeilenweise nach unten geblättert. Die nun oben nicht mehr auf dem Bildschirm sichtbare Zahlenzeile wird in die Zieldatei geschrieben.

Diese Datenstruktur wurde gewählt, weil sie leicht zu handhaben ist. Lediglich das Schreiben von Zeichen im Einfügemodus schafft ein kleines Problem. Wenn ein Zeichen eingefügt wird, müssen alle bereits vorhandenen dahinter stehenden Zeichen des Editierpuffers um ein Zeichen nach hinten geschoben werden. Wenn dies nicht möglich ist, weil der Puffer voll ist oder das letzte Zeichen nicht mehr auf den Bildschirm paßt, muß um eine Zahlenzeile nach unten geblättert werden. Dabei muß berücksichtigt werden, daß der Cursor nach dem Blättern an der richtigen Stelle steht.

Besondere Vorkehrungen müssen getroffen werden für den Fall, daß eine Zahlenzeile so groß ist, daß sie nicht auf einen Bildschirm paßt. Wenn in diesem Fall die letzte Bildschirmzeile gefüllt ist und ein weiteres Zeichen geschrieben (angefügt oder eingefügt) wird oder wenn der Cursor per Tastenfunktion <Nach rechts> oder <Nach unten> aus dem Bildschirm herauslaufen würde, muß die erste Bildschirmzeile in die Zieldatei geschrieben werden und vorn aus dem Editierpuffer gelöscht werden. Der restliche Inhalt des Editierpuffers muß um die Länge einer Bildschirmzeile nach vorn geschoben werden. Ebenso muß der Bildschirminhalt um eine Bildschirmzeile hochgeschoben werden.

Dritte Strategie:[10]

Das Programm soll leicht benutzbar sein und Hilfen zur korrekten Darstellung der Zahlen bieten. Jede Zahl wird, wenn sie geschrieben worden ist (wenn der Cursor die Zahl verläßt oder ein abschließendes Leerzeichen getippt wurde), auf Zulässigkeit überprüft. Unzulässige Zahlen sollen zu Fehlermeldung mit akustischem Signal und Korrekturaufforderung führen. Eine solche Aufforderung soll so lange erscheinen, bis die überprüfte Zahl zulässig oder gelöscht ist. Zahlen werden voneinander durch ein Leerzeichen getrennt, Zeilen durch das Trennzeichen „$“ und Seiten durch das Trennzeichen „_“.

Zahlenzeile ist Bildschirmzeile, jedenfalls soweit sie in eine Bildschirmzeile paßt, der Rest bleibt unsichtbar. Wenn der Cursor den rechten Rand erreicht und die Zahlenzeile noch nicht zu Ende ist, wird die Zeile um ein Zeichen nach links verschoben. Das vormals erste Zeichen wird dann nicht mehr dargestellt (Horizontal Scrolling).

Es gibt ein Funktionsmenü mit folgenden Funktionen: <Neue Datei anlegen>, <Datei öffnen>, <Datei speichern>, <Bearbeitung beenden>, <Einfügemodus einschalten>, <Einfügemodus ausschalten>, <Suchen>, <Ersetzen>. Der Cursor kann mit den üblichen Tasten gesteuert werden: <Nach rechts>, <Nach links>, <Nach unten>, <Nach oben>, <Pos1>, <Ende>, <Bild nach unten> und <Bild nach oben>. Die beiden letzten Tastenfunktionen wirken auf die Bildschirmseite, nicht auf die Zahlenseite. Durch <SHIFT><Bild nach unten> bzw. <SHIFT><Bild nach oben> werden Zahlenseiten geblättert.

Das Editieren soll in einem großen Puffer geschehen. Dafür wird ein Array von Zeichen (char) verwendet, das so groß ist, daß es gerade noch mit einem Zeiger vom Typ WORD indiziert werden kann. Dann werden während des Editierens „wenige“ Dateioperationen notwendig sein. In vielen Fällen werden alle Zahlen in diesen Puffer passen. Einfügen und Löschen kann durch den Pascalbefehl „move“ durchgeführt werden.

Der Editierpuffer wird aus einer Quelldatei (leer bei Ersterfassung) gefüllt und in eine Zieldatei entleert. Der noch nicht gelesene Teil der Quelldatei liegt sozusagen „hinter“, der bereits geschriebene Teil der Zieldatei „vor“ dem Editierpuffer. Natürlich kann nicht der gesamte Editierpuffer auf dem Bildschirm erscheinen. Durch geeignete Maßnahmen, z. B. Zeiger, wird festgehalten, welcher Teil gerade sichtbar ist.

Beim Zurückblättern kann es geschehen, daß Teile, welche nicht im Editierpuffer, wohl aber in der Zieldatei stehen, sichtbar werden müssen. In diesem Fall muß zunächst der gesamte Editierpuffer an die Zieldatei angehängt werden. Dann muß der gesamte noch nicht behandelte Teil der Quelldatei (eingelesen und ebenfalls)

[10] ebenfalls von Lothar Oppor

an die Zieldatei angehängt werden. Weiter muß die Zieldatei zur neuen Quelldatei werden. Die alte Quelldatei darf gelöscht werden. Eine neue, leere Zieldatei muß geschaffen werden. Dann muß dafür gesorgt werden, daß der Editierpuffer am besten in der Mitte die Zahlenzeilen enthält, auf die positioniert worden ist.

Es muß zwei verschiedene Arten von Syntaxprüfungen geben. Die einzelnen Tastendrücke müssen geprüft werden auf Menüoperationen (eingeleitet durch eine Funktionstaste, z. B. F3), Cursorsteuertaste oder Trennzeichen. Bei Eingabe eines Trennzeichens (Zahlen-, Zeilen- oder Seitentrennzeichen) muß die davor stehende Zahl auf Zulässigkeit geprüft werden. Mit der Anzahl der zulässigen Darstellungsformate für Zahlen steigt die Komplexität dieser Syntaxprüfung.

Anstelle des oben beschriebenen Editierpuffers mit sequentieller Struktur könnte auch eine dynamische vorwärts und rückwärts verkettete Liste von Elementen verwendet werden. Die Elemente sollten so groß sein, daß sie im Normalfall genau eine Zahl aufnehmen können (z. B. Länge 128 Zeichen, weil dies die Puffergröße von DOS ist). Damit sind Einfügungen eventuell einfacher und schneller zu erledigen. Zusätzlich muß der Fall, daß eine Zahl länger als ein Listenelement sein kann, bei der Syntaxprüfung berücksichtigt werden.

Als Datenstrukturen eignen sich serielle Zeichenketten oder verkettete Listen von Zeichenketten.
Mehrdimensionale Arrays sind wegen der festen Grenzen vermutlich nicht sehr vorteilhaft.

Editoraufbau:

Zum Schluß wollen wir noch kurz die Operationen zusammenstellen, die der Editor anbieten sollte.[11]

- Aufbau einer Editormaske,
- Menü-Steuerung,
- Methoden zum Laden von Datenteilen in die die Datenstruktur (evtl. mit Umwandlung des Datenformats),
- Methoden zum Speichern von Datenteilen in die Datei (evtl. mit Umwandlung des Datenformats),
- Zahlen-Eingaberoutine,
- Routine für die Ausgabe von Datenteilen auf dem Bildschirm,
- Cursor- oder Maus-Steuerung,
- Routinen zum Scrollen/Verschieben des aktuellen Datenteils am Bildschirm.

[11] In dieser Form wurden die Operationen von Ingo Schellhammer zusammengestellt.

5.3 Dreiecksflug

5.3.1 Aufgabenstellung

Franz Fürchtenichts möchte von Frankfurt nach Fulda, von Fulda weiter nach Friedberg und von Friedberg zurück nach Frankfurt fliegen. Die Distanz von Frankfurt nach Fulda beträgt 97 km, der Kurs ist 55 Grad. Von Fulda nach Friedberg sind es 68 km, der Kurs ist 250 Grad, und von Friedberg nach Frankfurt sind es 36 km bei Kurs 205 Grad. Das Flugzeug fliegt mit einer Reisegeschwindigkeit von 160 km/h.

Bevor Franz in Frankfurt startet, hört er im aktuellen Wetterbericht, daß ein Westwind aus 270 Grad mit 15 km/h weht.

Schreiben Sie ein Programm, das für beliebige Dreiecke nach Eingabe von Distanzen, Kursen, Windrichtung und -stärke die Steuerkurse und die voraussichtlichen Flugzeiten ausgibt. Berechnen Sie damit für jede der drei Teilstrecken Frankfurt – Fulda – Friedberg – Frankfurt den Kurs, den Franz steuern muß, um das geplante Dreieck zu fliegen, und die Flugzeit, mit der er rechnen muß.

Rechnen Sie mindestens **drei** weitere Beispiele.

5.3.2 Lösungsidee

In der Fliegerei ist (wie bei der Seefahrt) 0° im Norden, 90° im Osten usw. Das ergibt sich auch aus der Lage der drei angegebenen Städte zueinander und der Angabe, daß ein Westwind aus 270° weht. (Ein Westwind weht *aus* Westen, nicht *nach* Westen.) Bei zu großen Windgeschwindigkeiten sollte der Flug nicht erlaubt werden. 160 km/h sind nicht sehr viel, wenn man gegen einen Orkan mit Windgeschwindigkeiten bis zu 120 km/h ankämpfen muß!

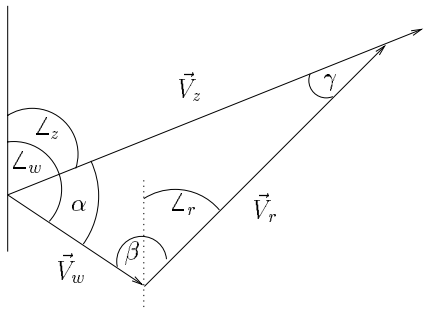
Bei der Aufgabe handelt es sich um eine Standardaufgabe des Typs: Gegeben zwei Seiten und ein angrenzender Winkel eines Dreiecks, berechne die dritte Seite und die übrigen Winkel.

5.3.3 Berechnungsverfahren

Zur Berechnung der Steuerkurse werden trigonometrische Funktionen und die Vektorrechnung verwendet. Dabei idealisieren wir die Situation, denn wir gehen — wie in der Aufgabenstellung angegeben — von konstanten Geschwindigkeiten aus. Außerdem berücksichtigen wir weder Auf- und Abstiegsphasen noch die Erdkrümmung, d. h. wir betrachten die Erdoberfläche in den für den jeweiligen

Flug interessierenden Ausschnitten als Ebene, was für weit auseinanderliegende Städte natürlich nicht mehr stimmt.

Auf das Flugzeug wirken gleichzeitig zwei Kräfte ein, zum einen der Flugzeugantrieb, zum anderen der Wind. Diese beiden Kräfte werden als Vektoren betrachtet, deren Betrag gerade die entsprechende Geschwindigkeit ist, die Richtung ist der jeweilige Kurs. Wir haben es insgesamt mit drei Vektoren zu tun, siehe die folgende Skizze.



$\vec{V}_w$: Windvektor
$\vec{V}_r$: Reisevektor
$\vec{V}_z$: Zielvektor
$\angle_w$: Windrichtung in Grad (umgerechnet in die Richtung, in die der Wind weht)
$\angle_r$: Steuerkurs in Grad
$\angle_z$: Zielrichtung in Grad

Gegeben: $\vec{V}_w$ (also V_w und $\angle_w$), V_r und $\angle_z$
Gesucht: V_z, $\angle_r$

Man trägt von demselben Punkt der Vertikalen aus den Windvektor und eine Halbgerade in Richtung des Zielvektors ab. Im Endpunkt des Windvektors ist der noch unbekannte Reisevektor anzusetzen. Um seinen Endpunkt zu finden, schlägt man um den Endpunkt des Windvektors einen Kreis, dessen Radius der Betrag des Reisevektors ist. Der Schnittpunkt von Kreis und Halbgerade ist Endpunkt des Reisevektors und des Zielvektors.

Weht der Wind genau von hinten oder von vorne (d. h. es gilt Windkurs = Zielkurs oder Windkurs = Zielkurs $\pm 180°$), so ist eine Sonderbehandlung erforderlich. Andernfalls kann bei den weiter unten aufgeführten Formeln eine Division durch 0 eintreten. Diese Sonderbehandlung ist einfach, denn in diesen Fällen können Reise- und Windgeschwindigkeit einfach addiert bzw. subtrahiert werden.

Sei r die Reisegeschwindigkeit, w die Windgeschwindigkeit und z die tatsächliche Geschwindigkeit des Flugzeugs.

Zunächst ermittelt man die Reiserichtung $\angle_r$. Nach dem Sinussatz gilt

$$\frac{\sin\gamma}{\sin\alpha} = \frac{w}{r}, \quad \text{also} \quad \gamma = \arcsin\left(\frac{w\cdot\sin\alpha}{r}\right),$$

und mit

$$\alpha = \angle_w - \angle_z \quad \text{folgt} \quad \gamma = \arcsin\left(\frac{w\cdot\sin(\angle_w - \angle_z)}{r}\right).$$

Wegen

$$\beta - \angle_r = 180^\circ - \angle_w$$

und

$$\alpha + \beta + \gamma = 180^\circ$$

(Wechselwinkel- und Winkelsummensatz) folgt

$$\begin{aligned}\angle_r &= \angle_w + \beta - 180^\circ \\ &= \angle_w - \alpha - \gamma \\ &= \angle_z - \gamma,\end{aligned}$$

also

$$\angle_r = \angle_z - \arcsin\left(\frac{w\cdot\sin(\angle_w - \angle_z)}{r}\right).$$

Damit läßt sich die tatsächliche Geschwindigkeit z des Flugzeuges ermitteln:

$$\begin{aligned}\frac{z}{r} &= \frac{\sin\beta}{\sin\alpha} = \frac{\sin(180^\circ - \alpha - \gamma)}{\sin\alpha}, \quad \text{also} \\ z &= r\cdot\frac{\sin(180^\circ - \angle_w + \angle_z + \angle_r - \angle_z)}{\sin(\angle_w - \angle_z)} \\ &= r\cdot\frac{\sin(180^\circ - \angle_w + \angle_r)}{\sin(\angle_w - \angle_z)}.\end{aligned}$$

Die Flugzeit ergibt sich nun als Quotient von Entfernung und der tatsächlichen Geschwindigkeit.

Für die Strecke von Frankfurt nach Fulda des Beispiels erhält man (mit $w = 15$ km/h, $r = 160$ km/h, $\angle_w = 90^\circ$ und $\angle_z = 55^\circ$) als Steuerkurs

$$\angle_r = 55^\circ - \arcsin\left(\frac{15\cdot\sin(35^\circ)}{160}\right) = 51{,}9^\circ$$

und als tatsächliche Geschwindigkeit

$$z = 160 \text{ km/h} \cdot \frac{\sin(180° - 90° + 51,9°)}{\sin(35°)} = 172 \text{ km/h}$$

Die Reisezeit beträgt

$$\frac{97 \text{ km}}{z} = 0,564 \text{ Stunden} \approx 34 \text{ Minuten.}$$

Entsprechend erhält man für die 2. Teilstrecke den Steuerkurs $251,8°$, eine tatsächliche Geschwindigkeit von 146,1 km/h und die Flugdauer 28 Minuten. Für die 3. Teilstrecke ergeben sich ein Steuerkurs von $209,9°$, 153 km/h als tatsächliche Geschwindigkeit und die Flugdauer 14 Minuten. Damit beträgt die gesamte Flugzeit 1 Stunde 16 Minuten.

5.3.4 Halbformale Programmbeschreibung

Da Turbo Pascal keine arcsin-Funktion zur Verfügung stellt, muß man sich diese selbst programmieren unter Verwendung der vorhandenen arctan-Funktion und der Umformungsregel $\tan\alpha = \frac{\sin\alpha}{\sqrt{1-\sin^2\alpha}}$.

Mit den angegebenen Formeln kann nun leicht ein Programm erstellt werden, das die gewünschten Daten berechnet. Ein möglicher Ablauf ist folgender:

```
erfrage die Reisegeschwindigkeit;
erfrage die Windgeschwindigkeit;
erfrage die Windrichtung;
drehe den Vektor der Windrichtung um 180°;
erfrage Kurs und Entfernung des 1. Ortes;
berechne den Steuerkurs;
berechne die benötigte Zeit;
gib Steuerkurs und benötigte Zeit aus;
erfrage Kurs und Entfernung des 2. Ortes;
berechne wie zuvor den Steuerkurs;
berechne wie zuvor die benötigte Zeit;
gib Steuerkurs und benötigte Zeit aus;
berechne momentane x- und y-Koordinate des Flugzeugs;
berechne Kurs zum Landeplatz;
berechne Entfernung zum Landeplatz;
gib Kurs und Entfernung zum Landeplatz aus;
berechne den Steuerkurs;
```

```
berechne die benötigte Zeit;
gib Steuerkurs und benötigte Zeit aus;
berechne Gesamtzeit;
gib Gesamtzeit aus.
```

Hier wird der dritte Kurs zum Ansteuern des Ausgangsflugplatzes nicht mehr erfragt, sondern anhand der aktuellen Position des Flugzeugs vom Programm ermittelt. Damit werden Ungenauigkeiten in der Eingabe der Distanzen vermieden.

5.3.5 Erweiterungen

Es liegt nahe, nicht nur Dreieckskurse zu fliegen, sondern auch Kurse mit mehreren Zwischenstopps. Dazu ist es nur notwendig, zunächst die Anzahl n der Stopps zu erfragen und dann die Kursberechnung in einer n-mal durchlaufenen Schleife durchzuführen.

Die Windgeschwindigkeit und -richtung kann von Teilstrecke zu Teilstrecke variieren. Das erfordert nur für jeden Teilflug das Einsetzen anderer Werte in die bereits bekannten Formeln.

Man kann die dritte Dimension berücksichtigen, die ja bisher vernachlässigt wurden, indem für unterschiedliche Flughöhen auch unterschiedliche Windstärken und eventuell auch unterschiedliche Windrichtungen angegeben werden. Dann muß die günstigste Flughöhe ermittelt werden. Aber auch das bringt nichts wesentlich Neues in das Programm, da die ermittelten Formeln gültig bleiben.

5.4 Kuriositäten und besonders nette Formulierungen

- Die Zahlen werden *auf dem Bildschirm* als Text eingegeben.
- assign(**g**, filename): ordnet der Dateivariablen **f** eine externe Datei filename zu.
- ...säuft viel Speicher
- geerdete Zeiger
- Vektorprogramm
- gebitmapte (*bitgemapt wäre auch hübsch ...*)
- FIFO-Stack — FILO-Stack (*Ein Stack ist* immer *LIFO, FIFO ist eine Schlange, FILO ist kein gängiger Begriff.*)
- Auch spricht alles für Modularisierung, da nichts dagegen spricht.
- Welche Probleme gibt es bei der Realisierung zu lösen:
  1. Eigene bei der Programmierung.
  2. Die Probleme, die der Anwender später hat, wenn man sie nicht löst.
- Der Ursprung entspricht dem Start- oder Wegpunkt, das Ziel liegt halt durch Strecke und Winkel festgelegt irgendwo anders.
- Eine Möglichkeit wäre, folgende Datenstruktur zu benutzen, die es erlaubt, unendlich viele Seiten und Spalten zu benutzen. (*Das wäre sehr erfreulich ...*)
- Zu dieser Gleichung gibt es zwei Möglichkeiten der Unlösbarkeit.
- Da die Ausgabe der Berechnungen über den Drucker an Lpt1 erfolgt, ist es günstig, einen solchen zu besitzen.
- ...etwas Licht in die frühmittelalterliche Kunst des Programmierens mit meinem spätkambrischen BASIC zu bringen.