GYMNASIUM GRAFING

KOLLEGSTUFE 1996 / 1998

Leistungskurs Mathematik M_1

Facharbeit: 16. Bundeswettbewerb Informatik 1997, erste Runde

---

Verfasser: Christoph Moder
Kursleiterin: StDin. Christine Vorbach

Abgabetermin: 2.2.1998

Bewertung: Note: _____ Punkte: _____

Unterschrift der Kursleiterin: _____

Inhaltsverzeichnis

Einleitung: Was ist der Bundeswettbewerb Informatik?

Programmieren macht Spass! Es ist zwar manchmal nervenaufreibend, wenn man mühsam an einem Programm schreibt und sich dann beim Probelauf ein Fehler zeigt, dessen Ursache man einfach nicht findet, man startet das Programm wieder und wieder, überwacht Variablen, setzt Breakpoints, aber plötzlich ist der Computer schon wieder abgestürzt. Jedes Programm ist eine neue Herausforderung, und jedesmal lernt man etwas dazu. Und eine weitere Herausforderung ist der Bundeswettbewerb Informatik, bei dem ich jetzt zum zweiten Mal teilgenommen habe.

Er wurde von der „Gesellschaft für Informatik e.V." (GI) und Prof. Dr. Volker Claus im Jahre 1980 gegründet, der auch die ersten Wettbewerbe, die ähnlich wie „Jugend forscht" organisiert waren, durchführte. Seit dem Wettbewerb 1985/86 ist der Bundeswettbewerb Informatik ein staatlich anerkannter bundesweiter Schülerwettbewerb und wird von den Kultusministerien der Länder unterstützt, d.h. er hat den gleichen Status wie die sicherlich bekannteren Bundeswettbewerbe Mathematik, Physik und Chemie. Finanziert wird er von der GI und der GMD („Gesellschaft für Mathematik und Datenverarbeitung mbH") sowie vom Bundesministerium für Bildung und Wissenschaft.

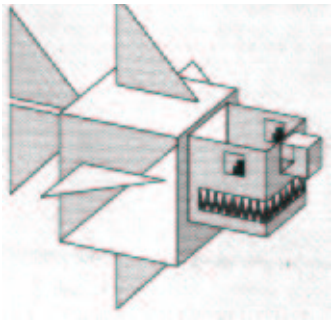
Der Wettbewerb soll bei den Schülern das Interesse an der Informatik wecken, sie sollen sich mit ihr und ihrer zunehmenden gesellschaftlichen Bedeutung beschäftigen; dazu tragen auch die phantasievollen Aufgaben (wie z.B. über Königin Eleonore, die in Farnsworth Castle belagert wird) und das aufwendig gestaltete Aufgabenblatt (s.u.: zwei Bilder vom Aufgabenblatt) bei. So soll natürlich gleichzeitig für die Informatik geworben und ihre gesellschaftliche Akzeptanz gefördert werden, und die Jugendlichen sollen auch generell ein größeres Verständnis für analytische und konstruktiv−ingenieursmäßige Vorgehensweisen bekommen. Schließlich möchten die Veranstalter auch indirekt die Informatik−Ausbildung in der Schule verbessern, indem sie vorschlagen, anhand der Wettbewerbsaufgaben mit den Schülern die Prinzipien der Informatik durchzuarbeiten oder die Schüler mit Hilfe dieser Aufgaben zu überprüfen. Ich stelle mir vor, dass das etwa so ablaufen könnte wie der „Mathe Pluskurs", der am Gymnasium Grafing von Herrn Herbert Langer veranstaltet wird. Dort werden vor allem Aufgaben von alten Mathematik−Bundeswettbewerben besprochen und durchgerechnet, mit dem Ergebnis, dass die Schüler des Gymnasiums Grafing sehr erfolgreich beim Bundeswettbewerb Mathematik teilnehmen. Es hilft den Schülern sicherlich, wenn ihnen Techniken wie z.B. Rekursion an konkreten Beispielen (wie z.B. der Aufgabe 3 „Rekursive Besen") erklärt wird, statt mit theoretischen Modellen, bei denen sie keinen praktischen Nutzen erkennen.

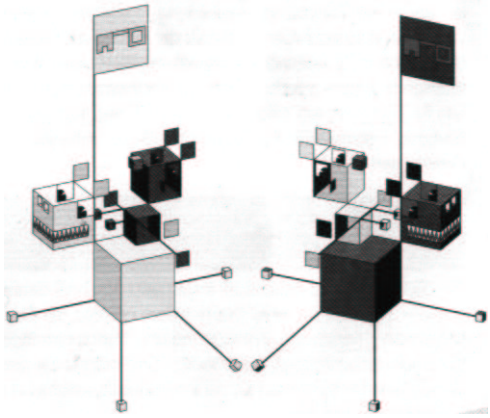
Der Bundeswettbewerb Informatik dauert jeweils ca. ein Jahr und umfasst drei Runden. In der ersten Runde, die im September beginnt, gibt es fünf Aufgaben, die anschaulich und nicht allzu schwer sind. (Die Aufgabenblätter werden in einer Auflage von ca. 100000 Stück gedruckt und an alle Schulen in Deutschland mit Sekundarstufe I und II und an ehemalige Teilnehmer − wie mich − geschickt.) Außerdem ist Gruppenarbeit gestattet und auch erwünscht, damit die Hürden für Informatik−Neulinge nicht zu groß sind und möglichst viele Leute zum Mitmachen bewegt werden können. Wer in der ersten Runde, egal ob allein oder im Team, von den maximal erreichbaren 25 Punkten (5 pro Aufgabe) 12 bekommen hat, darf bereits in der zweiten Runde teilnehmen, bei der aber keine Gruppenarbeit mehr erlaubt ist. Dort gibt es drei Aufgaben, die entsprechend schwieriger sind; insbesondere sind deutlich mehr Informatik−Vorkenntnisse nötig. Außerdem gibt es keine Höchstpunktzahl mehr, d.h. es reicht nicht aus, die Aufgaben „nur" zu lösen, sondern man muss seine Lösung darüberhinaus noch erweitern und verbessern, um Zusatzpunkte zu erhalten. Dabei werden vor allem Verbesserungen der Algorithmen und intelligente Lösungen belohnt, während einfache „Kosmetik" am Programm (wie zahlreiche zusätzliche Funktionen und Spielereien) und am Mensch−Maschine−Dialog kaum Zusatzpunkte bringt. Diese sind aber notwendig, denn nur die ca. 30 Besten der zweiten Runde kommen in die dritte und letzte Runde, die in Form eines mehrtägigen Kolloquiums bei einer Firma ausgerichtet wird, die die Runde finanziert. Die Teilnehmer bearbeiten dort in kleinen Gruppen von ca. fünf Leuten sehr schwierige Probleme, für die meistens keine angemessenen Techniken oder Tricks bekannt sind; allein eine gute Vorbereitung reicht also nicht aus! Die Vorgehensweisen und Ergebnisse werden danach zusammen besprochen, und außerdem gibt es halbstündige Einzelgespräche mit Fachleuten, die sich so ein detailliertes Bild über die Fähigkeiten der Teilnehmer machen können. Sie ermitteln etwa 6 Bundessieger, die dann in die Studienstiftung des deutschen Volkes aufgenommen werden, außerdem werden jene vier Leute ausgesucht, die Deutschland bei der Internationalen Olympiade in Informatik vertreten. Schließlich wird mit allen Teilnehmern der dritten Runde ein mehrtägiges Seminar zu einem Spezialthema der Informatik und ihrer Anwendungen durchgeführt.

Mitmachen darf jeder bis zu einem Alter von 21 Jahren, soweit er seine Ausbildung noch nicht abgeschlossen hat bzw. einen Beruf ausübt und, falls er nicht die deutsche Staatsbürgerschaft hat, während des Wettbewerbs in Deutschland wohnt oder eine staatlich anerkannte deutsche Schule besucht. Erlaubt sind alle höheren Programmiersprachen, das heißt Assembler und Maschinenbefehle wie Peeks und Pokes sind verboten, weil sie z.T. Insiderwissen voraussetzen; so wäre eine Chancengleichheit nicht mehr gegeben. Generell sind aber die Aufgaben so gestellt, dass Kenntnisse über die Technik und Hardware nicht viel weiterhelfen und die Programme mit fast jeder Programmiersprache auf fast jedem Computer geschrieben werden können. (Beim 16. Wettbewerb, den diese Facharbeit behandelt, hat ein Teilnehmer die Aufgabe 3 „Rekursive Besen" in Postscript programmiert (siehe „Lösungshinweise und häufige Fehler"). Postscript ist eigentlich keine Programmiersprache, sondern eine

Druckersprache, mit der der Computer dem Drucker die zu druckenden Daten übermittelt. Der Drucker decodiert dann diese Befehle mit Hilfe seines Postscript−Interpreters und druckt entsprechend. Eigentlich läuft also das Programm dieses Teilnehmers nicht im PC, sondern im Drucker ab!)

Wie gesagt: der Bundeswettbewerb Informatik ist eine Herausforderung; er bietet interessante und, vor allem in der zweiten Runde, auch praxisorientierte Aufgaben. Zur Lösung der Aufgaben kommt man auf neue Ideen, und, für mich ganz wichtig: Mathe−Leistungskursler können ihn als Facharbeit einbringen!
Mein Fazit: Der BW Informatik erfordert zwar viel Arbeit, aber er hat mir auch Spass gemacht!!!





Aufgabe 1: Belagerung um Farnsworth Castle

Aufgabenstellung:

AD 1314. Nicht mehr lange können die Ritter Königin Eleonores auf Farnsworth Castle die Angreifer des Blythlethwick−Clans abhalten. Die einzige Hoffnung der Königin ist die Benachrichtigung ihres Sohnes John, der sich mit Julee, der Prinzessin des Clans, verlobt hat. Wäre sie auf Farnsworth, müßten die Blythlethwicks verhandeln. Doch die Königin kann sich nicht sicher sein, daß ihr Bote nicht abgefangen wird. Würde ihr Plan bekannt, wäre Blutvergießen unvermeidbar.
Eleonore schickt darum zwei Boten. Einen mit einer verschlüsselten Botschaft, den anderen mit einem Schlüssel. Bei der Verschlüsselung geht sie folgendermaßen vor:
Sie schneidet eine quadratische Schablone aus Leder, die in quadratische Felder eingeteilt ist. Manche dieser Felder sind ausgeschnitten. Diese Schablone legt die Königin auf ein Stück Papier und schreibt die ersten Buchstaben ihrer Botschaft von links nach rechts, oben nach unten, durch die ausgeschnittenen Felder. Dann dreht sie die Schablone um 90 Grad im Uhrzeigersinn, schreibt weiter und wiederholt den Vorgang noch zwei weitere Male.
Die Nachricht ist:

```
KOMM UND
BRING JULEE NACH
FARNSWORTH
```

Schablone und verschlüsselte Nachricht sehen so aus:

```
BRKOAE
ER IMN
NMS WN
OGA CJ
UHRNUD
 L FTH
```

1. Welchen Anforderungen müssen Anzahl und Anordnung der Löcher genügen, damit die Schablone zur Verschlüsselung geeignet ist?
2. Gib ein einfaches Verfahren für den Entwurf von geeigneten Schablonen an.
3. Schreibe ein Programm, das anhand einer gegebenen Schablone eine Nachricht ver− und entschlüsseln kann. Wie sieht die mit der obigen Schablone verschlüsselte Fassung folgender Rückantwort aus?

```
JULEE
MIT STALLKNECHT
DURCHGEBRANNT
```

1. Anforderungen an Anzahl und Anordnung der Löcher

Da die Königin die Schablone dreimal dreht, d.h. die Schablone in 4 verschiedenen Richtungen verwendet, und dabei jedesmal durch jedes Loch genau einen Buchstaben schreibt, schreibt sie somit insgesamt vier Buchstaben pro Loch. Da die verschlüsselte Nachricht genausoviele Buchstaben (und Leerzeichen) enthält, wie die Schablone Quadratfelder besitzt, ergeben sich daraus folgende Anforderungen an die Schablone:

1. Sie muss mindestens soviele nutzbare Quadratfelder besitzen, wie die nicht verschlüsselte Nachricht Buchstaben (inklusive Leerzeichen) hat (nutzbares Quadratfeld: ein Feld, auf dem sich ein Loch befinden darf).
2. Weil mit jedem Loch 4 Buchstaben geschrieben werden, dürfen maximal ¼ der nutzbaren Felder der Schablone Löcher sein, damit in der verschlüsselten Nachricht nicht mehrere Buchstaben übereinander geschrieben werden. (die Beispielschablone hat 36 Felder, davon sind 36 / 4 = 9 Felder Löcher).
3. Die Löcher müssen so angeordnet sein, dass kein Feld unter der Schablone mehrfach beschrieben wird, d.h. dort, wo sich bei einem bestimmten Drehwinkel gegenüber der ursprünglichen Lage der Schablone (0° oder 90° oder 180° oder 270°) ein Loch befindet, darf sich bei den drei anderen Drehwinkeln keines befinden (auch daraus folgt, dass bei der Schablone max. ¼ der nutzbaren Felder Löcher sein dürfen). Beispiel: wenn man die Beispielschablone um 90° im Uhrzeigersinn dreht, ist in ihrer linken oberen Ecke ein Loch. Wenn man sie stattdessen in der Ausgangsposition (=0°) beläßt, ist an der linken oberen Ecke kein Loch, genausowenig bei einer Drehung gegenüber der Ausgangsposition um 180° oder 270°.
4. Bei einer Schablone mit ungeradzahliger Seitenlänge gilt zusätzlich: Das mittlere Feld darf kein Loch sein: es wird nämlich bei Drehung der Schablone immer auf sich selbst abgebildet und kann somit Anforderung Nr. 3 nicht erfüllen. Also ist bei Schablonen mit ungerader Seitenlänge die Anzahl der nutzbaren Felder immer um 1 geringer als die Gesamtanzahl der Felder. Beispiele:

– Eine Schablone mit 25 Feldern kann Nachrichten der maximalen Länge 25 − 1 = 24 Buchstaben verschlüsseln (siehe 1.).

– Die maximale Anzahl der Löcher muss, wie oben erwähnt, aus der Anzahl der nutzbaren Felder berechnet werden; also gilt z.B. für eine Schablone mit mit der Seitenlänge 5 Felder:
Gesamtanzahl der Felder: 25; davon nutzbar: 24; => maximal 24 / 4 = 6 Löcher!

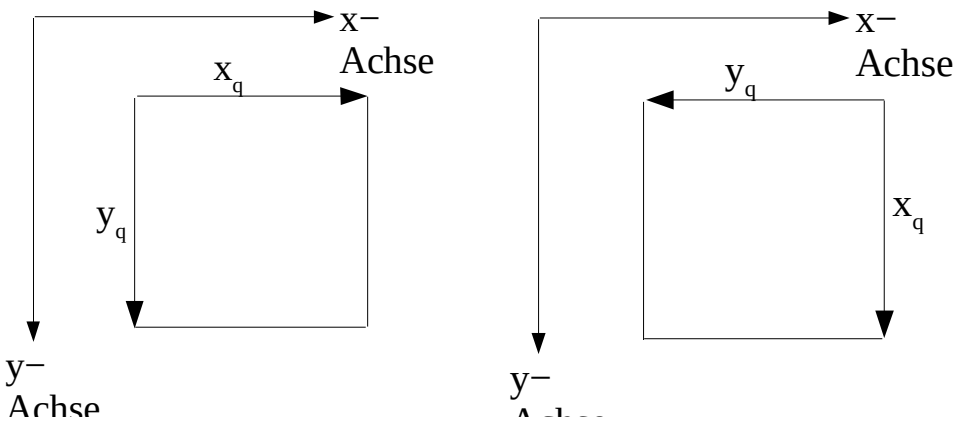
2. Geeignetes Verfahren zur Erstellung von Schablonen

Zuerst muss die benötigte Größe der Schablone bekannt sein. Dazu ermittelt man die Länge der Nachricht (in Buchstaben, inkl. Leerzeichen) und verwendet eine Schablone, deren Anzahl an quadratischen Feldern die nächstgrößere Quadratzahl ist. Anders gesagt: die Seitenlänge der Schablone muss die auf die nächste natürliche Zahl aufgerundete Wurzel aus der Länge der Nachricht sein, z.B. die Nachricht hat 17 Buchstaben => die Schablone muss eine Seitenlänge von Wurzel aus 17 = 4,1231... , also (aufgerundet) 5 Feldern haben, d.h. $5^2 = 25$ quadratische Felder besitzen. (Ausnahmefälle gibt es bei Schablonen mit ungeradzahliger Seitenlänge; genaueres siehe Programmdokumentation.) Danach müssen die Positionen der Löcher bestimmt werden. Aus den oben genannten Beschränkungen ergibt sich, dass bestimmte Positionen „verboten“ sind. Also wäre ein zweckmäßiges Vorgehen, die Position jedes Lochs aus den noch freien, nicht „verbotenen“ Feldern auszusuchen:

1. bei Schablonen mit ungeradzahliger Seitenlänge: das mittlere Feld als „verboten“ markieren
2. für jedes Loch ein Feld aussuchen, wo sich weder bereits ein Loch befindet, noch das als „verboten“ markiert ist; anschließend die 3 Felder, die sich bei Drehung der Schablone um 90°, 180° und 270° an der Position des Loches befinden, als „verboten“ markieren.

3. Schreiben eines Ver− und Entschlüsselungsprogramms

Lösungsidee:



Wenn man ein Quadrat um 90° im Uhrzeigersinn dreht, dann ist die Seite, die vorher oben war, rechts, und die Seite, die vorher links war, oben. Wenn man nun auf das Quadrat zwei Vektoren, nämlich x_q und y_q, setzt, die anfangs parallel zur x− bzw. y−Achse liegen und die x− bzw. y−Koordinate eines beliebigen Punkts im Quadrat repräsentieren (siehe 1. Bild), dann sieht die Situation nach einer Drehung des Quadrats um 90° im Uhrzeigersinn folgendermaßen aus (s. 2. Bild):

– x_q, der vorher in Richtung der x−Achse zeigte, liegt nun parallel zur y−Achse, und zeigt auch in diese Richtung. Also gilt:
Die x−Koordinate jedes Punkts im Quadrat wird nach einer 90°−Drehung zu seiner neuen y−Koordinate.

– y_q, der vorher in Richtung der y−Achse zeigte, liegt nun parallel zur x−Achse, aber zeigt entgegengesetzt. Also gilt hier:
Die y−Koordinate jedes Punkts im Quadrat wird nach einer 90°−Drehung zu seiner „gespiegelten" neuen x−Koordinate, d.h. neue x−Koordinate = maximale Seitenkoordinate − alte y−Koordinate (maximale Seitenkoordinate bedeutet: bei mit 0 beginnender Zählung der Seitenlänge die höchste Koordinate; also bei einer 5x5−Schablone bedeutet das: Koordinaten von 0 bis 4 => neue x−Koordinate = 4 − alte y−Koordinate).

Das ist eigentlich die komplette Lösungsidee, da man für eine Drehung um 180° diesen Schritt lediglich noch einmal durchführen muss, für eine 270°−Drehung entsprechend insgesamt dreimal usw.

Programmdokumentation:

Zuerst verlangt das Programm die Eingabe, ob die Nachricht ver− oder entschlüsselt werden soll (gespeichert in *Nachricht_verschluesseln*: 1 bei Verschlüsseln, 0 bei Entschlüsseln). Dann folgt die Eingabe der Nachricht in die Variable *Nachricht_vorher*. Je nachdem, ob die Nachricht ver− oder entschlüsselt werden soll, wird sie als normaler Satz (mit *gets()* eingelesen) oder (mit Hilfe der Funktion *lies_Quadrat_ein()*) als verschlüsselte Nachricht in Quadratform eingelesen (dazu gibt der Benutzer die verschlüsselte Botschaft Zeile für Zeile ein und gibt nach der letzten Zeile nichts mehr ein, sondern drückt noch einmal die Eingabetaste). Aus der Buchstabenanzahl dieser Nachricht wird die Größe der benötigten Schablone berechnet; ihre Seitenlänge wird in der Variable *Seitenlaenge* gespeichert (*Seitenlaenge* = die auf die nächste natürliche Zahl aufgerundete Wurzel aus der Länge der Nachricht). Dabei gibt es eine Ausnahme: wenn die Länge der zu verschlüsselnde Nachricht genau eine ungerade Quadratzahl ist (die Schablone also eine ebenfalls ungerade Seitenlänge hat), dann muss eine Schablone der nächstgrößeren Seitenlänge benutzt werden, da bei Schablonen mit ungeradzahliger Seitenlänge ein Feld nicht benutzt werden kann (s.o.: z.B. die Nachricht hat 25 Buchstaben => eine Schablone mit der Seitenlänge 5 Felder kann aber nur 24 Buchstaben verschlüsseln => man benötigt (mindestens) eine Schablone der Seitenlänge 6). Beim Entschlüsseln spielt diese Tatsache keine Rolle, da (falls die verschlüsselte Nachricht nicht falsch eingegeben wurde) die Nachrichtenlänge immer exakt der Schablonengröße entspricht und bei Nachrichten mit ungeradzahliger Länge das Zeichen in der Mitte des Quadrats kein Buchstabe sein kann (sondern ein Leerzeichen sein muss; das folgt aus dem Verschlüsselungsprozess, weil eine geeignete Schablone in der Mitte nie ein Loch haben darf). Die Anzahl der nutzbaren Felder ist gespeichert in der Variablen *nutzbare_Felder*. Sie berechnet sich als das Quadrat der Seitenlänge, bei ungerader Seitenlänge wird noch 1 abgezogen (s.o.). Die Anzahl der Löcher in der Schablone ist

entsprechend ¼ von *nutzbare_Felder*; anschließend muss der Benutzer in einer Schleife die Koordinaten der Löcher in der Schablone eingeben (immer im Format Zeile/Spalte, ohne Leerzeichen o.ä., z.B. 1/3 für das erste Loch der Beispielschablone). Danach folgt das Kernstück des Programms: eine Schleife, in der viermal je ein Viertel der Nachricht ver− bzw. entschlüsselt und danach die Schablone um jeweils 90° dreht und die Koordinaten sortiert. Danach wird die ver− bzw. entschlüsselte Nachricht ausgegeben (die entschlüsselte Nachricht wird als ganz normaler String mit Hilfe von *printf()* ausgegeben, während die verschlüsselte Nachricht mit Hilfe der Funktion *gib_Quadrat_aus()* in der quadratischen Form der Schablone hingeschrieben wird).
Das Drehen der Schablone erfolgt in der Funktion *drehe_Koordinate_90_Grad* wie oben beschrieben: die Spalte des Lochs wird zu seiner neuen Zeile, und die neue Spalte ist der Maximalwert der Zeilen− bzw. Spaltenkoordinate minus der alten Zeilenkoordinate. Da die Zählung der Zeilen− und Spaltenkoordinaten im Programm bei 0 beginnt, errechnet sich dieser Maximalwert als *Seitenlaenge* −1 (z.B. die Seitenlänge ist 6: also gehen die Spalten− und Zeilenkoordinaten jeweils von 0 bis 5, und 5 = 6 − 1).
Da die Reihenfolge der Elemente des Arrays *Loch* die Reihenfolge angibt, in der die Löcher vom Programm benutzt werden, die Königin aber nach dem Drehen der Schablone nicht ebenfalls um 90° gedreht weiterschreibt, sondern wieder oben anfängt, müssen also die Elemente des Arrays *Loch* neu sortiert werden, und zwar so, dass, je weiter oben ein Loch in der gedrehten Schablone ist, desto weiter vorne wird es im Array einsortiert. Das Sortieren geschieht in der Funktion *sortiere_Koordinaten* und funktioniert folgendermaßen: in einer Schleife wird, vom ersten Loch im Array angefangen bis zum letzen Loch, zuerst in einer weiteren Schleife von den im Array dahinterliegenden Lochkoordinaten jenes Loch herausgesucht, das weiter oben in der Schablone liegt, als das untersuchte Loch, und mit ihm vertauscht (wenn es kein weiter oben liegendes Loch gibt, wird keine Vertauschung unternommen) (dies entspricht dem Algorithmus *Selection Sort*[1]). Dabei werden die Koordinaten der Löcher in Positionen relativ zum Anfang der Schablone umgewandelt (nach der Formel Position = Zeile * Seitenlänge der Schablone + Spalte); dadurch ist der Vergleich der Reihenfolge zweier Löcher sehr einfach.

Halbformale Programmbeschreibung:

Lies alle notwendigen Angaben ein

Wiederhole 4 mal:

    Wiederhole mit jedem Loch auf der Schablone
        zum Verschlüsseln: ordne den aktuellen Buchstaben der entschlüsselten Nachricht an die durch die Position des Lochs gegebene Position in der verschlüsselten Nachricht;
        zum Entschlüsseln: hänge den Buchstaben in der verschlüsselten Nachricht dessen Position durch die Position des Lochs gegeben ist, an die entschlüsselte Nachricht dran;
    Schleifenende

    drehe die Schablone um 90°, indem die Koordinaten jedes Lochs umgerechnet werden
    sortiere die Koordinaten der Schablone

Schleifenende
Gib die verschlüsselte bzw. entschlüsselte Nachricht aus

Programmcode:

```
#include <stdio.h>    // printf()
#include <string.h>   // strlen()
#include <ctype.h>    // toupper()
#include <conio.h>    // getche()
#include <math.h>     // sqrt()

#define MAX_LOECHER   20
#define MAX_NACHRICHT MAX_LOECHER*4+1       // für jedes Loch 4 Felder + Mittelfeld

// wenn man eine Zahl durch 2 teilt & es gibt einen Rest, dann ist sie ungerade
#define ungerade(x)   (x%2!=0)

typedef struct
       {
        int Zeile;
        int Spalte;
       } TLoch;

int Seitenlaenge=0;          // die Seitenlänge der Schablone in Feldern
TLoch Loch[MAX_LOECHER];     // ein Array mit den Koordinaten der Löcher
int Anzahl_Loecher=0;        // die Anzahl der Löcher

// die Nachricht vor dem Ver−/Entschlüsselungsvorgang
```

1 Aus dem Buch „Algorithmen in C++“ von Robert Sedgewick, Addison−Wesley−Verlag, S. 125

```
char Nachricht_vorher[MAX_NACHRICHT+1]="";

// die Nachricht nach dem Ver-/Entschlüsselungsvorgang
char Nachricht_nachher[MAX_NACHRICHT+1]="";

void sortiere_Koordinaten()
{
 int i,j;
 TLoch temp;
 int oberstes;

 // alle Löcher der Reihenfolge nach durcharbeiten
 for(i=0;i<Anzahl_Loecher;i++)
 {
  // davon ausgehen, dass das jeweils bearbeitete Loch bereits an seinem
  // richtigen Platz ist, also von den im Array dahinterliegenden Löchern das
  // oberste ist
  oberstes=i;

  // finde von den im Array dahinterliegenden Löchern das Loch, das noch weiter
  // oben liegt: alle Löcher dahinter überprüfen, wenn eines gefunden wurde,
  // das weiter oben ist, beide vertauschen
  for(j=i;j<Anzahl_Loecher;j++)
  {
   if(Loch[j].Zeile*Seitenlaenge+Loch[j].Spalte<
        Loch[oberstes].Zeile*Seitenlaenge+Loch[oberstes].Spalte)
               oberstes=j;
  }

  if(oberstes!=i)     // wurde eines gefunden, das noch weiter oben liegt?
  {
   // ja => beide Löcher vertauschen
   temp.Zeile=Loch[i].Zeile;
   temp.Spalte=Loch[i].Spalte;

   Loch[i].Zeile=Loch[oberstes].Zeile;
   Loch[i].Spalte=Loch[oberstes].Spalte;

   Loch[oberstes].Zeile=temp.Zeile;
   Loch[oberstes].Spalte=temp.Spalte;
  }

 }
}

void drehe_Koordinate_90_Grad(TLoch *l)
{
 int temp;

 temp=l->Zeile;
 l->Zeile=l->Spalte;
 l->Spalte=Seitenlaenge-temp-1;
}

void lies_Quadrat_ein()
{
 int i=1;
 char Eingabe_String[MAX_NACHRICHT+1]="";

 printf("Gib die zu entschlüsselnde Nachricht Zeile für Zeile ein;\n"
        "drück nach der letzten Zeile einfach nochmal die Eingabetaste:\n");

 do
 {
  printf("%d. Zeile: ",i++);
  gets(Eingabe_String);       // lies die Zeile ein...

  // ...und häng sie an die vorigen Zeilen dran
  strcat(Nachricht_vorher,Eingabe_String);

  // lies solange ein, bis eine leere Zeile eingegeben wird
 }while(strlen(Eingabe_String)>0);
}

void gib_Quadrat_aus(int Laenge)
{
 int i,j;

 for(i=0;i<Laenge;i++)               // für jede Zeile der Nachricht...
 {
  // für jede Spalte der Nachricht einen Buchstaben schreiben
  for(j=0;j<Laenge;j++) putchar(Nachricht_nachher[i*Laenge+j]);
  // am Ende der Zeile einen Zeilenumbruch
  putchar('\n');
 }
}

// Hauptprogramm
int main()
{
 int nutzbare_Felder=0;
 char Eingabe='v';    // nur f. Eingabe, ob ver- oder entschlüsselt werden soll
```

```
int Nachricht_verschluesseln=1;     // verschlüsseln oder entschlüsseln ?
int i,j;             // Schleifenzählvariablen

printf("Soll eine Nachricht (v)erschlüsselt oder (e)ntschlüsselt werden? ");
Eingabe=getche();
if(toupper(Eingabe)=='E') Nachricht_verschluesseln=0;
printf("\n");

if(Nachricht_verschluesseln==1)
{
 printf("Zu verschlüsselnde Nachricht:\n");
 gets(Nachricht_vorher);
}
else
{
 lies_Quadrat_ein();
}

if(strlen(Nachricht_vorher)>MAX_NACHRICHT)
{
 printf("Nachricht für das Programm zu lang.\nBeende das Programm.\n");
 return 1;
}

// Seitenlänge = nächstgrößere natürliche Zahl als die Wurzel aus
// der Nachrichtenlänge
Seitenlaenge=ceil(sqrt(strlen(Nachricht_vorher)));

// wenn eine Nachricht verschlüsselt werden soll und die Nachricht exakt so
// viele Buchstaben hat, wie eine Schablone mit ungerader Seitenlänge Felder
// hat, muss die nächstgrößere Schablone benutzt werden (s. Text)
if((Nachricht_verschluesseln==1)&&ungerade(Seitenlaenge)&&
       (strlen(Nachricht_vorher)==Seitenlaenge*Seitenlaenge))
{
 Seitenlaenge++;
}

printf("\nDie Nachricht hat %d Buchstaben => "
       "benutze eine Schablone der Seitenlänge %d.\n",
       strlen(Nachricht_vorher),Seitenlaenge);

nutzbare_Felder=Seitenlaenge*Seitenlaenge;
if(ungerade(Seitenlaenge)) nutzbare_Felder--;

Anzahl_Loecher=nutzbare_Felder/4;

printf("\nEingabe der %d Löcher im Format Zeile/Spalte (z.B. 2/3):\n",
       Anzahl_Loecher);

for(i=0;i<Anzahl_Loecher;i++)
{
 printf("%d. Loch: ",i+1);
 scanf("%d/%d",&Loch[i].Zeile,&Loch[i].Spalte);

 // wenn das Loch außerhalb der Schablone liegt oder, bei ungerader Seiten-
 // länge, das mittlere Feld als Loch verwendet werden soll, dann Fehlermeldung
 if((Loch[i].Zeile<1)||(Loch[i].Spalte<1)||
       (Loch[i].Zeile>Seitenlaenge)||(Loch[i].Zeile>Seitenlaenge)||
       (ungerade(Seitenlaenge)&&(Loch[i].Zeile==Seitenlaenge/2+1)&&
       (Loch[i].Spalte==Seitenlaenge/2+1)))
 {
  printf("Falsche Eingabe! Da kann kein Loch sein! Nochmal:\n");
  i--;
  continue;
 }

 Loch[i].Zeile--;Loch[i].Spalte--;  // Arrays beginnen bei 0
}

sortiere_Koordinaten();

printf("\n\n");

// die eingegebene Nachricht mit Leerzeichen auffüllen
if(strlen(Nachricht_vorher)<nutzbare_Felder)
{
 for(i=strlen(Nachricht_vorher);i<nutzbare_Felder;i++) Nachricht_vorher[i]=' ';
 // am Schluss eine 0 zur Terminierung des Strings
 Nachricht_vorher[nutzbare_Felder]='\x0';
}

// den Ausgabestring erstmal auf Leerzeichen setzen
memset(Nachricht_nachher,' ',Seitenlaenge*Seitenlaenge);
// am Schluss eine 0 zur Terminierung des Strings
Nachricht_nachher[Seitenlaenge*Seitenlaenge]='\x0';

/*********************\
* Ver-/Entschlüsseln: *
\*********************/
for(i=0;i<4;i++)     // insgesamt wird die Schablone in 4 Positionen benutzt
{
 if(Nachricht_verschluesseln==1)    // wenn verschlüsselt werden soll
```

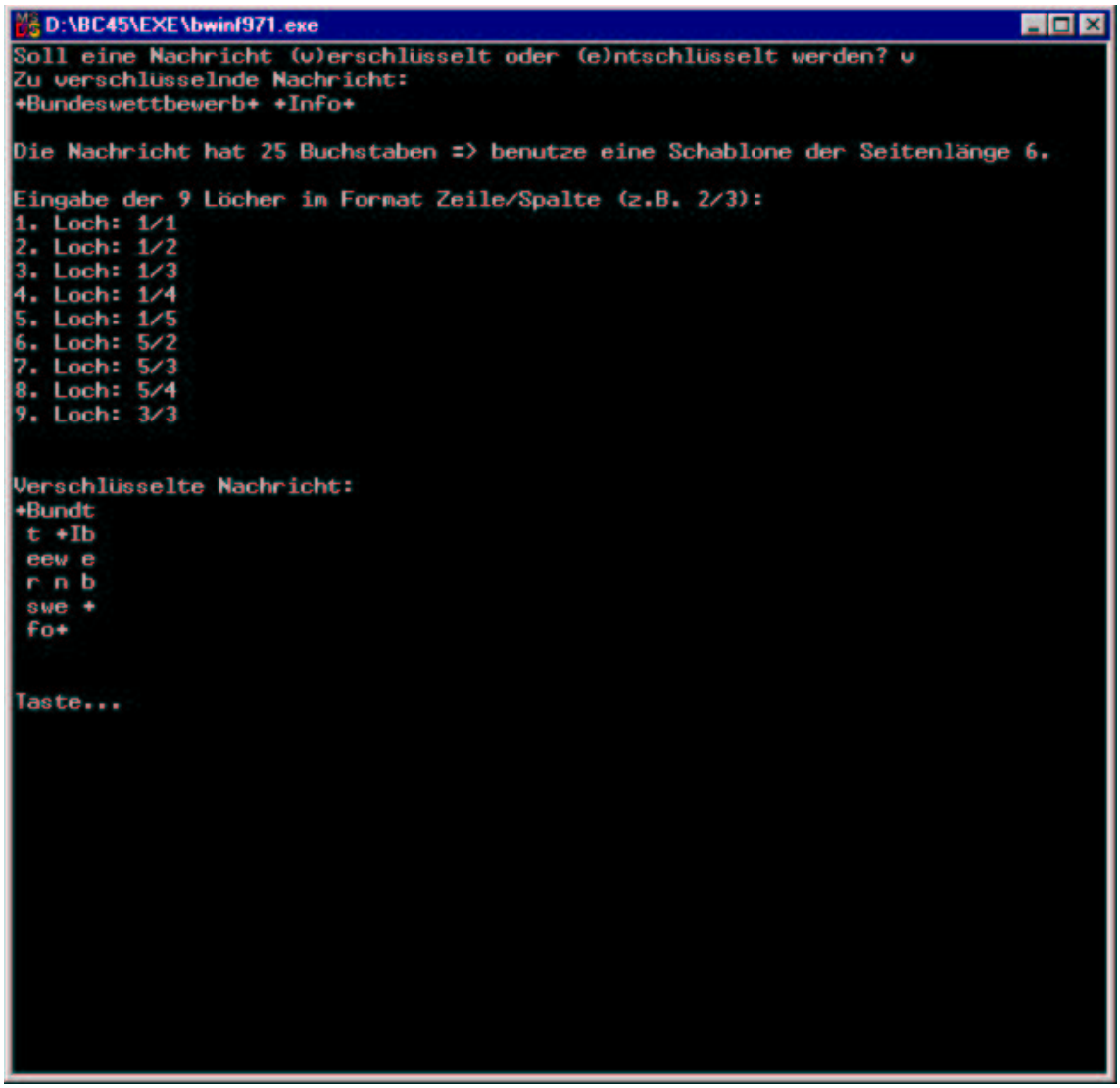
```
  {
   // die Buchstaben der nicht verschlüsselten Nachricht an die durch die Löcher
   // angegebenen Positionen der verschlüsselten Nachricht setzen
   for(j=0;j<Anzahl_Loecher;j++)
   {
    Nachricht_nachher[Loch[j].Zeile*Seitenlaenge+Loch[j].Spalte]=
        Nachricht_vorher[i*Anzahl_Loecher+j];
   }

   // Schablone um 90° drehen
   for(j=0;j<Anzahl_Loecher;j++) drehe_Koordinate_90_Grad(&Loch[j]);
   sortiere_Koordinaten();
  }
  else         // wenn entschlüsselt werden soll
  {
   // alle Buchstaben der verschlüsselten Nachricht unter den Löchern an
   // die entschlüsselte Nachricht anhängen
   for(j=0;j<Anzahl_Loecher;j++)
   {
    Nachricht_nachher[i*Anzahl_Loecher+j]=
        Nachricht_vorher[Loch[j].Zeile*Seitenlaenge+Loch[j].Spalte];
   }

   // Schablone um 90° drehen
   for(j=0;j<Anzahl_Loecher;j++) drehe_Koordinate_90_Grad(&Loch[j]);
   sortiere_Koordinaten();
  }
 }

 /*************************\
 * Ausgabe des Ergebnisses: *
 \*************************/
 if(Nachricht_verschluesseln==1)     // wenn verschlüsselt wurde
 {
  printf("Verschlüsselte Nachricht:\n");
  gib_Quadrat_aus(Seitenlaenge);
 }
 else                               // wenn entschlüsselt wurde
 {
  printf("Entschlüsselte Nachricht: \"%s\"",Nachricht_nachher);
 }

 printf("\n\nTaste...\n");
 getch();

 return 0;
}
```

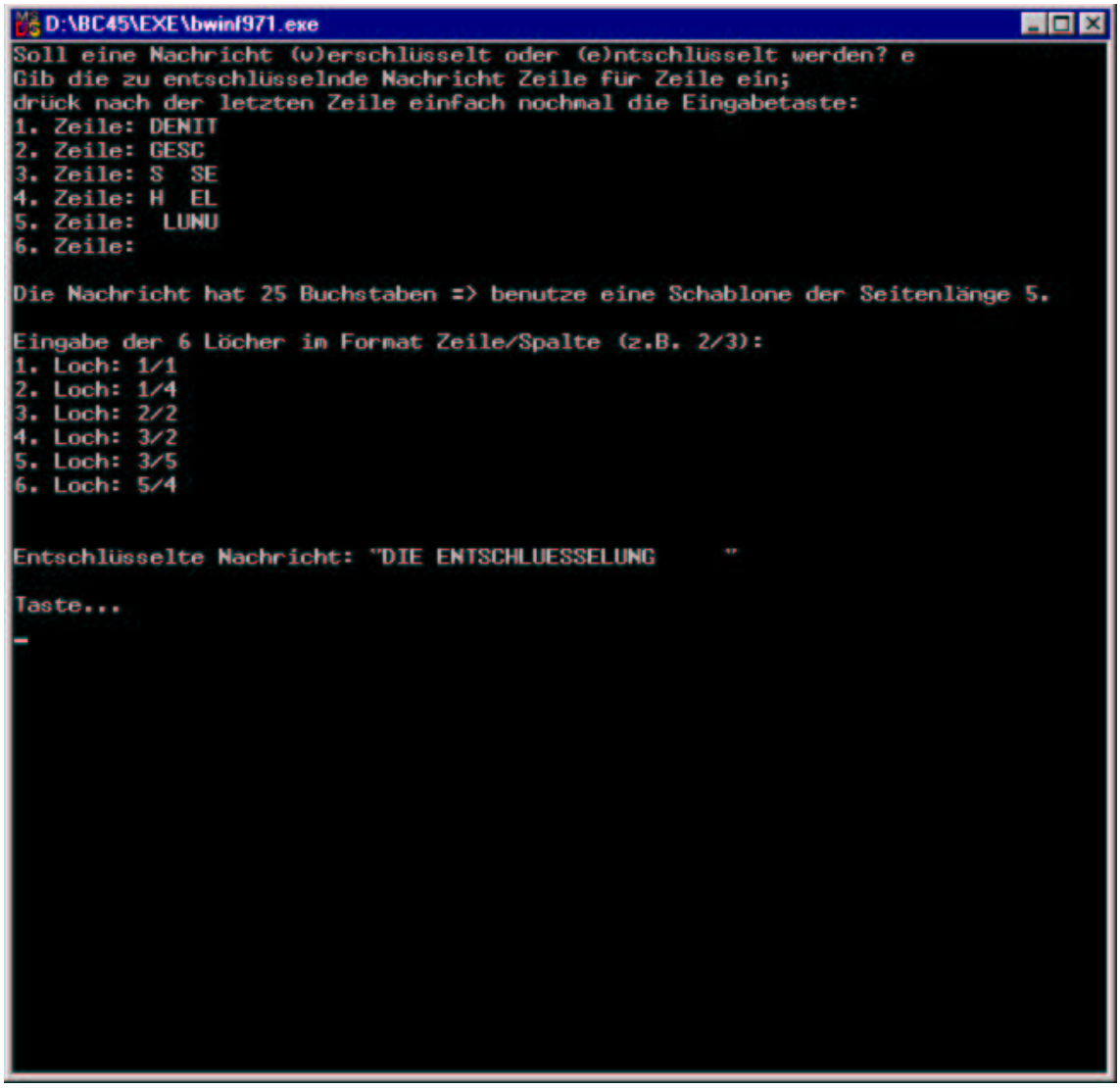
Anmerkungen:

- die maximale Anzahl der Löcher wird durch die Konstante *MAX_LOECHER* bestimmt; diese Konstante hat den Wert 20, was bedeutet, dass es maximal 20 Löcher geben darf, die Nachricht also maximal 4 * 20 = 80 Zeichen lang sein darf;

- Bei der Eingabe der verschlüsselten Nachricht wird nicht überprüft, ob die eingegebenen Zeilen der verschlüsselten Nachricht die richtige Länge haben oder ob zu viele / zu wenige Zeilen eingegeben wurden; darauf muss der Benutzer achten. Bei Eingabe der verschlüsselten Nachricht müssen alle Felder eingegeben werden, leere Felder sind durch Leerzeichen einzugeben. Auch dass die Schablone „stimmt“ (s.o.), muss der Benutzer gewährleisten.
- bei Eingabe der Loch−Koordinaten ist darauf zu achten, dass die Koordinatenpaare nur durch einen Schrägstrich getrennt werden und sich keine Leerzeichen o.a. dazwischen befinden;
- es wäre theoretisch möglich, zum Ver− / Entschlüsseln eine Schablone zu benutzen, die eine größere Seitenlänge als benötigt hat; in diesem Fall ließe sich auch die Anzahl der Löcher reduzieren, so dass die Anzahl der Löcher in der Schablone dann kleiner als ¼ der Anzahl der nutzbaren Felder ist; ich habe diese Fälle aber vernachlässigt, da sie zusätzliche Benutzereingaben erfordert hätten, den Code unübersichtlicher gemacht hätten und schließlich in der Aufgabenstellung nicht ausdrücklich verlangt werden;

Programmablauf−Protokoll (Screenshots und Lösungen):




D:\BC45\EXE\bwinf971.exe
Soll eine Nachricht (v)erschlüsselt oder (e)ntschlüsselt werden? e
Gib die zu entschlüsselnde Nachricht Zeile für Zeile ein;
drück nach der letzten Zeile einfach nochmal die Eingabetaste:
1. Zeile: DENIT
2. Zeile: GESC
3. Zeile: S SE
4. Zeile: H EL
5. Zeile: LUNU
6. Zeile:
Die Nachricht hat 25 Buchstaben => benutze eine Schablone der Seitenlänge 5.
Eingabe der 6 Löcher im Format Zeile/Spalte (z.B. 2/3):
1. Loch: 1/1
2. Loch: 1/4
3. Loch: 2/2
4. Loch: 3/2
5. Loch: 3/5
6. Loch: 5/4
Entschlüsselte Nachricht: "DIE ENTSCHLUESSELUNG   "
Taste...

Aufgabe 2: Partyvorbereitungen

Aufgabenstellung:

Katrin Käfer möchte ein großes Fest feiern. Dafür sind etliche Aufgaben zu erledigen: Salate machen, Pizza vorbereiten, Zimmer ausräumen, Nachbarn warnen und dergleichen mehr. Katrin möchte nicht die ganze Arbeit alleine machen und überlegt sich daher, einige ihrer Freundinnen und Freunde zur Mitarbeit heranzuziehen. Nun stellt sich natürlich die Frage, wie viele Personen sie um Mithilfe bitten soll.

Etwas formaler stellt sich ihr Problem wie folgt dar: Es sind n Aufgaben zu erledigen. Die einzelnen Aufgaben sind nicht mehr teilbar. Für die i−te Aufgabe ist die Zeit t_i zu veranschlagen. Katrin allein würde also die Zeit $t_1 + t_2 + ... + t_n$ benötigen. Jede Person, die Katrin helfen könnte, kann nur einen Nachmittag, d.h. nicht mehr als 5 Stunden Zeit aufbringen.

Um die minimale Anzahl Helfer zu berechnen, müßte Katrin alle Möglichkeiten, Kombinationen von Aufgaben auf Helfer zu verteilen, ausprobieren. Bei den vielen zu erledigenden Aufgaben dauert ihr das aber zu lange.

Aufgabe:
Entwirf eine Strategie, die folgendes leistet:
− Sie ermittelt nicht die optimale Lösung durch Probieren aller Kombinationen.
− Sie ordnet nicht einfach jeder Person genau eine Aufgabe zu.
− Sie verplant höchstens doppelt so viele Personen, wie im besten Fall nötig gewesen wären.

Beschreibe Deine Strategie und beweise, daß Katrin damit nicht mehr als doppelt so viele Helfer einspannt, wie nötig gewesen wären.

Schreibe ein Programm, das Deine Strategie realisiert.

Gib für n = 9, $t_1 = t_2 =...= t_7$ = 45 min, t_8 = 165 min, t_9 = 105 min und pro Person zur Verfügung stehender Zeit 300 min an, zu welchem Ergebnis Deine Strategie führt (nicht die von Hand errechnete optimale Lösung!).
Wende Deine Strategie außerdem auf 10.000 mit dem Zufallszahlengenerator erzeugte Zahlen im Bereich 1 bis 20 und pro Person zur Verfügung stehender Zeit 25 an (n = 10.000, t_i Element aus {1,..., 20}) und gib an, wieviele Personen benötigt werden und wie lange sie im Durchschnitt arbeiten.

Beschreibung der Strategie:

Durch meine Strategie wird einer Person immer die Arbeit zugeteilt, die die noch freie Zeit dieser Person (freie Zeit = die insgesamt pro Person zur Verfügung stehende Zeit (im Beispiel 300 Minuten) minus die Dauer aller Arbeiten, die dieser Person bereits zugeordnet wurden) möglichst optimal ausfüllt − also von den Arbeiten, die maximal soviel Zeit benötigen, wie die freie Zeit der Person beträgt, wird diejenige ausgesucht, die am längsten dauert. Wird keine derartige Arbeit mehr gefunden, weil alle übrigen Arbeiten mehr Zeit beanspruchen, als die Person freie Zeit hat, wird eine weitere Person angestellt und die Arbeitszuteilung bei ihr nach dem gleichen Schema fortgesetzt.
Anders gesagt: Einer Person wird zuerst die am längsten dauernde Arbeit zugeteilt, da ihr noch keine Arbeiten zugeordnet wurden und sie somit noch keine Zeit verplant hat. Danach wird ihr also in mehreren Schritten jeweils die zeitaufwendigste Arbeit von den Arbeiten, die ihre noch zur Verfügung stehende Zeit nicht überschreitet, zugeordnet. Und zwar so lange, bis die freie Zeit der Person kürzer als die kürzeste noch nicht verteilte Arbeit ist. Dann beginnt das ganze mit der nächsten Person von vorne − insgesamt solange, bis alle Arbeiten verteilt sind.

Beweis, dass maximal doppelt so viele Leute wie eigentlich nötig verplant werden:

Es können folgende Fälle auftreten (die insgesamt pro Person zur Verfügung stehende Zeit wird im Folgenden „*Gesamtzeit*“ genannt; das Verhältnis von insgesamt einer Person zugeordneter Arbeit zur Gesamtzeit bezeichne ich als Auslastung *a* dieser Person):

1. Wenn einer Person (die bisher noch keine Arbeit aufgetragen bekommen hat) eine Aufgabe zugeteilt wird, die länger als die halbe Gesamtzeit dauert, ist diese Person bereits zu über 50% ausgelastet, ungeachtet dessen, dass sie evtl. noch Arbeit zugeteilt bekommen kann.
2. Falls die ihr zugeteilte Arbeit jedoch kürzer als die halbe Gesamtzeit dauert, können folgende Fälle auftreten:
    1. Sie kann noch eine oder mehrere Arbeit(en) mit einer Gesamtdauer von mindestens der halben Gesamtzeit zugeteilt bekommen und ist somit ebenfalls zu mindestens 50% ausgelastet.
    2. Es gibt keine Arbeiten mehr, die ihre Auslastung *a* auf mindestens 50% steigern würden (also bleibt $a < 50\%$). Dies tritt aber nur bei der letzten Person auf, da sonst die zur Verfügung stehenden Arbeiten immer zu Fall 1 oder Fall 2.1 führen. Dieser Fall tritt aber nur dann auf, wenn bei allen anderen Personen die Auslastung *a'* größer als 100% − *a* ist, also jede einzelne der anderen Personen weniger Arbeitszeit frei hat, als diese letzte Person Arbeit hat. Wenn man also aus der Auslastung der letzten Person und einer der übrigen Personen den Durchschnitt bildet, so kommt als Ergebnis heraus:
    Durchschnittsauslastung $a_{\text{Durchschnitt}}$ = (*a'* + *a*) / 2 = ((größer als (100% − *a*)) + *a*) / 2 = (größer als 100%) / 2 = größer als 50%; (wie oben erwähnt ist gleichzeitig die Auslastung aller anderen Personen auch > 50% !)
    3. Sonderfall: Es ist insgesamt so wenig Arbeit vorhanden, dass nur eine einzige Person benötigt wird, die außerdem weniger als 50% ausgelastet ist (z.B. die pro Person zur Verfügung stehende Zeit beträgt 300 Minuten, aber alle Arbeiten zusammengenommen dauern nur 100 Minuten). Das heißt, in diesem Fall ist die Auslastung der Person egal, weil dann auch die optimale Lösung genau eine Person erfordert, also die Bedingung von vornherein erfüllt ist, dass nicht mehr als doppelt so viele Personen als bei der optimalen Lösung benötigt werden dürfen.

Daraus läßt sich erkennen, dass (abgesehen vom Sonderfall 2.3) sowohl die Durchschnittsauslastung der letzten Person und einer weiteren Person (wie bei 2.2 beschrieben) immer mindestens 50% beträgt und dass außerdem alle anderen Personen ebenfalls immer zu mindestens 50% ausgelastet sind. Mindestens 50% Auslastung bedeutet jedoch nichts anderes, als dass höchstens doppelt so viele Personen eingespannt werden, als benötigt würden, wenn die Auslastung 100% wäre. Die Auslastung der Personen bei der optimalen Lösung kann aber höchstens 100% sein (ach nee...), somit werden sowohl hier als auch im Sonderfall 2.3 höchstens doppelt so viele Personen eingespannt, als einer optimalen Lösung nach nötig wären (quod erat demonstrandum).

Programm, das meine Strategie realisiert:

Lösungsidee: siehe Beschreibung meiner Strategie

Programmdokumentation:

Das Programm besitzt die Konstante *TESTLAUF*; wenn diese definiert ist, führt das Programm den in der Aufgabenstellung beschriebenen Testlauf aus mit *Gesamtzeit* = 25 Minuten, *n* = 10000 und
$1 <= t_i <= 20$. Wenn diese Konstante auskommentiert ist (wie im Sourcecode der Fall), dann kann der Benutzer diese Daten selbst eingeben. Die folgende Beschreibung gilt für den normalen Programmablauf (*TESTLAUF* auskommentiert).
Zuerst muss der Benutzer die Anzahl der zu erledigenden Arbeiten eingeben; gespeichert wird sie, wie in der Aufgabenstellung vorgegeben, in der Variablen *n*. Danach wird zu jeder Arbeit die dazu benötigte Zeit eingelesen und im Array *t* gespeichert und gleichzeitig die Zeiten aller Arbeiten t_i zusammengezählt und in der Variablen *durchschnittliche_Arbeitszeit* gespeichert. Dann wird die pro Person zur Verfügung stehende Zeit in die Variable *Gesamtzeit* eingelesen.
Die Arbeiten werden anschließend in einer Schleife an die Personen verteilt − bei jedem Schleifendurchgang eine Arbeit.
Die Variable *am_Besten_passende_Arbeit* wird auf den Wert *KEINE_GEFUNDEN* gesetzt; dann wird in einer weiteren Schleife die für die Person am Besten passende Arbeitszeit herausgefunden: alle Arbeiten werden durchsucht; wenn eine der Arbeiten noch nicht als *VERGEBEN* markiert ist und gleichzeitig höchstens so lange dauert, wie die Person, die gerade bearbeitet wird, noch Zeit übrig hat, dann wird *am_Besten passende_Arbeit* auf die Nummer dieser Arbeit gesetzt, wenn die vorher durch *am_Besten_passende_Arbeit* angegebene Arbeit kürzer dauert oder *am_Besten_passende_Arbeit* noch den Wert *KEINE_GEFUNDEN* hat (ein Beispiel: die Person hat noch 5 Minuten freie Zeit; die gerade untersuchte Arbeit Nr. 7 dauert 4 Minuten, und *am_Besten_passende_Arbeit* hat den Wert *KEINE_GEFUNDEN*. Also wird *am_Besten_passende_Arbeit* der Wert 7 zugewiesen. Dann wird Arbeit Nr. 12 gefunden, die ebenfalls noch nicht vergeben ist, aber 5 Minuten dauert: das passt besser. Also bekommt jetzt *am_Besten_passende_Arbeit* den Wert 12). Nachdem dann alle Arbeiten untersucht worden sind, bekommt die Person jetzt die als am besten passend ermittelte Arbeit: *Arbeit_an_Person_verteilt* wird um die Dauer der neuen Arbeit erhöht (also die Dauer der Arbeit mit der Nummer *am_Besten_passende_Arbeit* wird zu *Arbeit_an_Person_verteilt* dazuaddiert) und diese Arbeit erhält den Wert *VERGEBEN*. Wenn es aber keine passende Arbeit gibt, d.h. *am_Besten_passende_Arbeit* enthält nach Schleifenende noch den Wert *KEINE_GEFUNDEN*, dann muss eine neue Person eingestellt werden: *Anzahl_Personen* wird um 1 erhöht und *Arbeit_an_Person_verteilt* auf 0 gesetzt (es interessiert ja nicht, wie lange jede Person arbeitet, sondern nur, wie viele Personen man braucht) − aber dieser neuen Person noch keine Arbeit zugeteilt (deshalb der Befehl *i−−* in Zeile 82 im Programm: in diesem Schleifendurchgang ist die Anzahl der verteilten Arbeiten nicht gestiegen). Dann beginnt die Schleife wieder von vorne − so lange, bis alle Arbeiten verteilt sind.
Danach wird die Variable *durchschnittliche_Arbeitszeit*, die die Summe aller Arbeiten enthält, durch die Anzahl der benötigten Personen geteilt (jetzt hat man die Durchschnittsarbeitszeit), und schließlich die Anzahl der benötigten Personen, die durchschnittliche Arbeitszeit (in Minuten) und die durchschnittliche Auslastung (in Prozent) ausgegeben.
Wenn das Programm mit *#define TESTLAUF* kompiliert wird, dann fallen die Benutzereingaben weg und die Arbeitszeiten werden per Zufallsgenerator ermittelt (10000 Eingaben vom Benutzer sind auch etwas viel).

Halbformale Programmbeschreibung:

lies die Anzahl der Arbeiten ein
lies in einer Schleife zu jeder Arbeit ihre Dauer ein und summiere sie gleichzeitig
lies die pro Person zur Verfügung stehende Zeit ein

in einer Schleife:
    setze *am_Besten_passende_Arbeit* auf *KEINE_GEFUNDEN*

    in einer Schleife: mit allen Arbeiten...
        speichere von den Arbeiten, die noch nicht vergeben sind und höchstens so
    lange dauern, wie die Person noch Zeit hat, diejenige, die am längsten dauert
    Schleifenende

    wenn eine passende Arbeit gefunden wurde: addiere die Dauer der ermittelten Arbeit zur Arbeitszeit der Person und markiere die Arbeit als *VERGEBEN*
    sonst: stelle eine neue Person ein (d.h. setze ihre Arbeitszeit auf 0 und erhöhe die Anzahl der Personen um 1) und mach diesen Schleifendurchgang nochmal (jetzt findet sich sicher eine Arbeit − die Person hat ja noch

ihre volle Zeit zur Verfügung)
Schleifenende

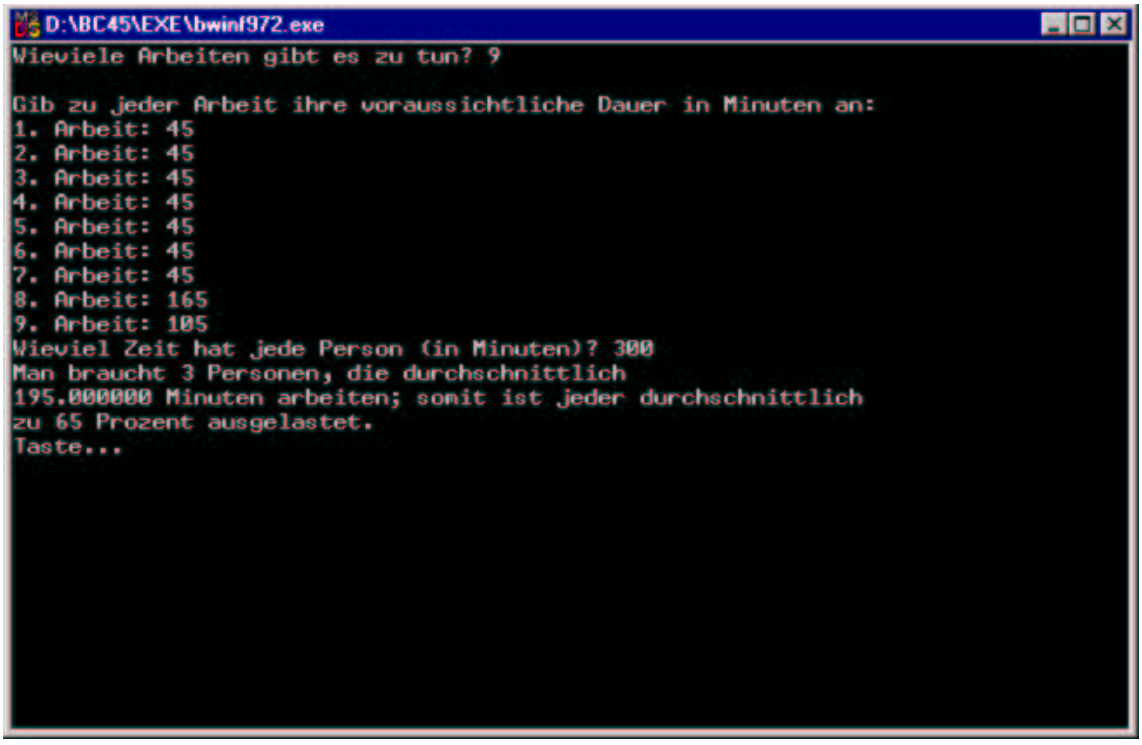
teile die aufsummierte Arbeitszeiten durch die Anzahl der Personen
gib die Anzahl der benötigten Personen, die durchschnittliche Arbeitszeit und die Auslastung aus

Programmcode:

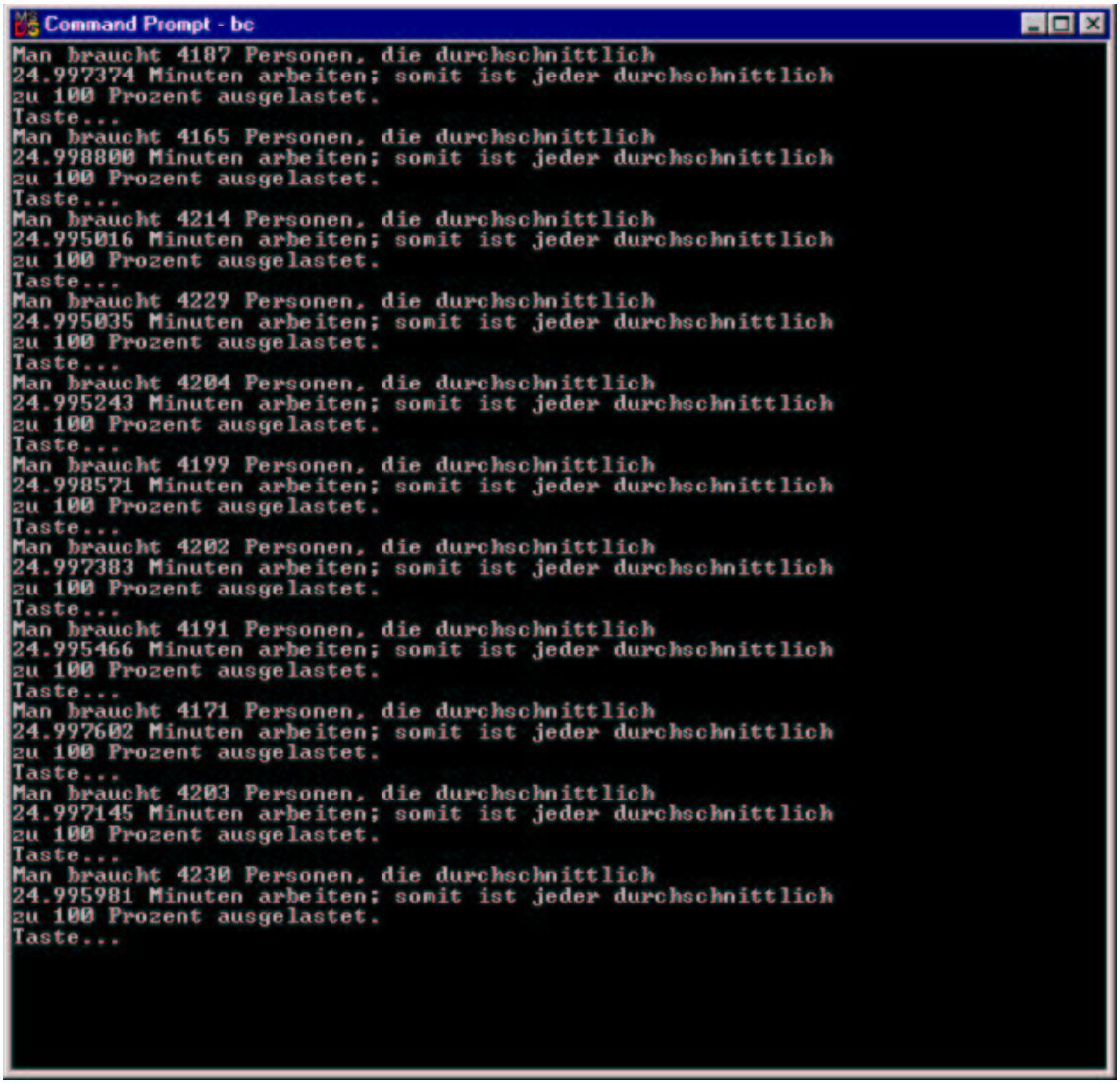
```
#include <stdio.h>    // printf()
#include <stdlib.h>   // random()
#include <conio.h>    // getch()

//#define TESTLAUF

#define MAX_ARBEITEN          10000
#define MAX_EINZELARBEIT      20

#define VERGEBEN              -1
#define KEINE_GEFUNDEN                -2

// Hauptprogramm
int main()
{
 int n;
 int t[MAX_ARBEITEN];
 int Anzahl_Personen=1;
 int Arbeit_an_Person_vergeben=0;           // soviele min. muss die Person schuften
 int am_Besten_passende_Arbeit;
 int Gesamtzeit=25;
 float durchschnittliche_Arbeitszeit=0;
 int i,j;                             // Schleifenzählvariablen

 #ifndef TESTLAUF
 printf("Wieviele Arbeiten gibt es zu tun? ");
 scanf("%d",&n);
 #else
 n=10000;
 #endif

 #ifndef TESTLAUF
 printf("\nGib zu jeder Arbeit ihre voraussichtliche Dauer in Minuten an:\n");
 #else
 randomize();
 #endif

 for(i=0;i<n;i++)
 {
  #ifndef TESTLAUF

  printf("%d. Arbeit: ",i+1);
  scanf("%d",&t[i]);

  #else
  t[i]=random(MAX_EINZELARBEIT)+1;
  #endif

  // die gesamte Arbeit aufsummieren (für die Statistik)
  durchschnittliche_Arbeitszeit+=t[i];
 }

 #ifndef TESTLAUF
 printf("Wieviel Zeit hat jede Person (in Minuten)? ");
 scanf("%d",&Gesamtzeit);
 #endif

 for(i=0;i<n;i++)
 {
  // suche am Besten passende Arbeit heraus:
  am_Besten_passende_Arbeit=KEINE_GEFUNDEN; // noch keine gefunden

  for(j=0;j<n;j++)
  {
   // wenn die Arbeit nicht bereits vergeben ist und die Arbeitszeit der Person
   // nicht übersteigt...
   if((t[j]!=VERGEBEN)&&(t[j]<=Gesamtzeit-Arbeit_an_Person_vergeben))
   {
    // wenn noch keine Arbeit ausgesucht, dann nimm die erstbeste
    if(am_Besten_passende_Arbeit==KEINE_GEFUNDEN) am_Besten_passende_Arbeit=j;
    // sonst: nimm sie, wenn sie länger dauert
    else if(t[j]>t[am_Besten_passende_Arbeit])
        am_Besten_passende_Arbeit=j;
   }
  }

  // keine passende Arbeit gefunden
  if(am_Besten_passende_Arbeit==KEINE_GEFUNDEN)
  {
   Anzahl_Personen++;
   Arbeit_an_Person_vergeben=0;
```

```
   i--; // hier wurde noch keine Arbeit vergeben
  }
  else
  {
   // Arbeit der Person zuordnen...
   Arbeit_an_Person_vergeben+=t[am_Besten_passende_Arbeit];
   // ...und Arbeit als vergeben deklarieren
   t[am_Besten_passende_Arbeit]=VERGEBEN;
  }
 }

 durchschnittliche_Arbeitszeit/=Anzahl_Personen;

 printf("Man braucht %d Personen, die durchschnittlich\n"
        "%f Minuten arbeiten; somit ist jeder durchschnittlich\n"
        "zu %.0f Prozent ausgelastet.\nTaste...\n",
        Anzahl_Personen,durchschnittliche_Arbeitszeit,
        durchschnittliche_Arbeitszeit*100/Gesamtzeit);

 getch();

 return 0;
}
```

Programmablaufprotokoll (Screenshots):



Der Programmablauf für die in der Aufgabenstellung angegebenen Daten; gleichzeitig ein Beweis, dass das Programm nicht die optimale Lösung liefert (die „von Hand“ errechnete optimale Lösung sieht so aus: 2 Personen, die durchschnittlich 292,5 Minuten arbeiten und somit zu 97,5% ausgelastet sind).



11 Testläufe (wie in der Aufgabe angegeben mit n=10000, *Gesamtzeit*=25 und $1<=t_i<=20$) (die 100% kommen daher, dass die Funktion *printf()* die Zahl auf die gewünschte Genauigkeit (keine Nachkommastelle) rundet).

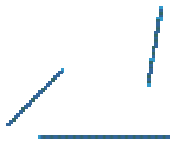
Anmerkungen:

- das Programm überprüft nicht die Eingaben des Benutzers; der Benutzer muss also selbst darauf achten, dass er keine ungültigen Werte eingibt (wie z.B. negative Arbeitszeiten oder Arbeiten, die länger dauern, als die Personen Zeit haben o.ä.), sondern für die Anzahl der Arbeiten, die Gesamtzeit und die Dauer jeder Arbeit natürliche Zahlen im Bereich von 1 bis 32767 benutzt
- die maximale Anzahl von Arbeiten, mit denen das Programm umgehen kann, ist durch die Konstante *MAX_ARBEITEN* auf 10000 Arbeiten festgelegt

Aufgabe 3: Rekursive Besen

Aufgabenstellung:

Heiner Huschelmutz, CAD−Spezialist, ist verzweifelt. Seine Frau war eine Woche auf Dienstreise und kommt morgen zurück. Huschelmutz möchte ihr die Wohnung in blitzblankem Zustand präsentieren; im Moment gleicht sie noch einem Schlachtfeld. Er will einen Besen holen, kann aber keinen finden; es ist nach Ladenschluß. Da kommt ihm eine Idee: In seiner Heimwerkstatt hat er eine computergetriebene Fräsmaschine. Warum soll sie ihm nicht einen Besen fräsen? Die Daten für den Besen sollen automatisch generiert werden. Zwecks Inspiration nimmt er drei Borsten eines Pinsels in die Hand, die er auf seinem Schreibtisch herumschiebt und anstarrt. Sie liegen so da:

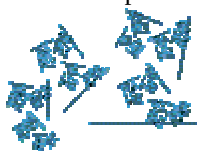


Heiner Huschelmutzs Sicht verschwimmt, und er sieht immer mehr Borsten, erst so:

dann so:

dann so:

Noch etwas später hat er folgendes Bild vor Augen:



Anfänglich ist Huschelmutz verwirrt, doch bald erkennt er ein Prinzip hinter dem Spiel, das seine Augen mit ihm treiben.

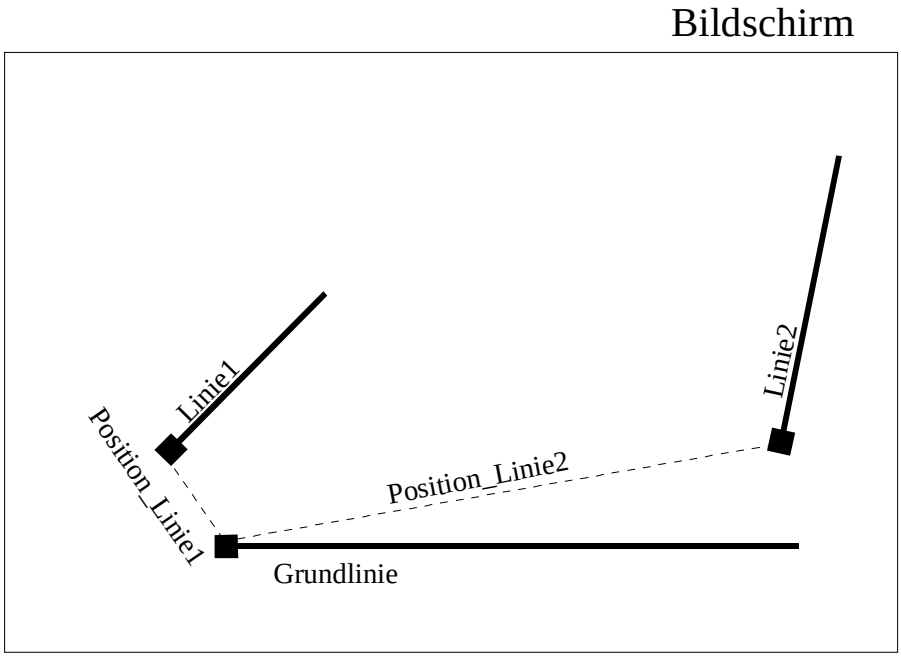
1. Nach welchem Prinzip ist Heiner Huschelmutzs Vision aus der ursprünglichen Borstenanordnung entstanden?
2. Schreibe ein Programm, welches das Prinzip über beliebig viele Schritte fortsetzt und jeweils die Borstenanordnung auf dem Bildschirm anzeigt. Schicke uns Bilder der ersten sechs Stufen.
3. Verändere die Ausgangslage der Borsten so, daß andere schöne Bilder entstehen. Schicke uns drei davon.

1. Prinzip von Heiner Huschelmutzs Vision

Das Bild ist ein Fraktal, d.h. es steckt (wenn man das Herstellungsprinzip über unendlich viele Stufen fortsetzen würde und entsprechend zoomen würde) in jedem Teil des Bildes, egal wie klein, das gesamte Originalbild.
Das Herstellungsprinzip heißt, wie im Aufgabentitel verraten, Rekursion. Konkret funktioniert das so: Eine Funktion zeichnet das Bild und ruft sich anschließend selbst auf. Auf der untersten Linie (im Programm und auch im Folgenden als *Grundlinie* bezeichnet) befinden sich zwei weitere Linien, die kürzer als die *Grundlinie* sind. Durch Rekursion wird erreicht, dass diese beiden kurzen Linien zu neuen *Grundlinie*n werden und auf jede von ihnen zwei weitere, entsprechend verkürzte Linien gesetzt werden. Diese werden wiederum zu *Grundlinien* und erhalten je zwei noch kleinere Linien, usw. Dabei wird aus jeder *Grundlinie* das Originalbild (mit 3 Linien) gemacht, bevor die kleinen Linie zu neuen *Grundlinien* werden. Dabei ist die verkleinerte Figur winkeltreu − also mathematisch ähnlich zum Originalbild. Um dies zu erreichen, braucht man eine Funktion, die an einer angegebenen Position auf einer angegebenen *Grundlinie* die beiden anderen, kurzen Linien zeichnet. Diese Funktion ruft sich danach zweimal selbst auf − für jede der kurzen Linien einmal − mit der jeweiligen kurzen Linie als neue *Grundlinie* und als Position die Stelle auf dem Bildschirm, an der sich die jeweilige *Grundlinie* befindet. Dieser Prozess könnte nun unendlich oft weitergehen, aber da weder unendlich viel Zeit noch ein unendlich großer Stack zur Verfügung steht und die Bildschirmauflösung auch ein begrenzender Faktor ist, läßt man die Funktion die Rekursionstiefe mitzählen und bei einer bestimmten Tiefe aufhören − auf dem Aufgabenblatt ist z.B. angegeben, dass man nur die ersten sechs Stufen angeben soll; danach kann die Rekursion aufhören.

2. Ein Programm für Heiner Huschelmutz

Lösungsidee:



Da auf jede der kleinen Linien wieder das gesamte Bild gezeichnet werden muss, gibt es für das Programm vor allem drei Sachen zu tun: erstens muss das Originalbild (die drei Linien) jedesmal verkleinert werden, zweitens, da diese kleinen Linien nicht unbedingt (auch im Beispiel nicht) die gleiche Neigung wie die *Grundlinie* haben, muss es auch gedreht werden, und drittens muss es dann an der richtigen Position am Bildschirm gezeichnet werden.
Das Verkleinern ist kein Problem: Da die *Grundlinie* auf eine der kleinen Linien abgebildet wird und die Länge dieser kleinen Linie bekannt ist, kann man den Faktor ausrechnen, um den die *Grundlinie* verkleinert wurde. Um den gleichen Faktor müssen auch die beiden kleinen Linien verkleinert werden, indem man ihre Längen mit dem Verkleinerungsfaktor (Skalierungsfaktor) multipliziert.
Wenn eine der beiden kleinen Linien (im Programm und auch im Folgenden *Linie1* genannt) gegenüber der *Grundlinie* um den Winkel α gedreht ist, und die andere kleine Linie (im Programm und auch im Folgenden *Linie2* genannt) gegenüber der *Grundlinie* um den Winkel β gedreht ist, dann gilt Folgendes: Wenn nun *Linie1* zur neuen *Grundlinie* wird, ist die neue *Linie1* gegenüber der neuen *Grundlinie* um den gleichen Winkel α gedreht (da das Bild ja winkeltreu verkleinert wird). Da die neue *Grundlinie* ebenfalls um den Winkel α gegenüber der Original*grundlinie*

gedreht ist, beträgt also die Drehung der neuen *Linie1* gegenüber der Original*grundlinie* $\alpha + \alpha$.. Und weil die Drehung der neuen *Linie2* gegenüber der neuen *Grundlinie* wieder β beträgt, ist sie gegenüber der Originalgrundlinie also um den Winkel $\alpha + \beta$ gedreht. Der Winkel einer kleinen Linie relativ zur Original*grundlinie* muss also immer zum (absoluten) Winkel der jeweiligen *Grundlinie* dazuaddiert werden, um den absoluten Winkel der kleinen Linie zu erhalten.
Die Position, an die die neuen Linien gezeichnet werden, d.h. die Anfangspunkte der neuen *Linie1* und der neuen *Linie2* lassen sich folgendermaßen bestimmen: Die Strecken vom Anfangspunkt der jeweiligen *Grundlinie* bis zu den Anfangspunkten von *Linie1* und *Linie2* (im Folgenden *Position_Linie1* und *Position_Linie2* genannt) werden genauso gedreht und skaliert wie die Linien *Linie1* und *Linie2* selbst. Sie beginnen aber beide am Anfangspunkt der jeweiligen *Grundlinie*; da dessen absolute Koordinaten bekannt sind, lassen sich damit auch die Endpunkte von *Position_Linie1* und *Position_Linie2* berechnen. Und an diesen Endpunkten beginnen die Linien *Linie1* und *Linie2*.
Insgesamt müssen dem Programm also folgende Daten bekannt sein:

- die (absolute) Position des Anfangspunkts der (Original−)*Grundlinie*, ihre Länge und ihr Winkel
- die Länge von *Linie1* und *Linie2* sowie deren Winkel relativ zur *Grundlinie*
- die Längen der Strecken vom Anfangspunkt der *Grundlinie* bis zu den Anfangspunkten der Linien *Linie1* und *Linie2* (also die Längen von *Position_Linie1* und *Position_Linie2*) sowie deren Winkel relativ zur Original−*Grundlinie*

Der rekursiv arbeitenden Funktion müssen folgende Daten übergeben werden:

- die Koordinaten des Anfangspunkts der neuen *Grundlinie* (absolut und in kartesischer Form)
- der Winkel (absolut) und die Länge der neuen *Grundlinie*

Daraus lassen sich alle anderen Koordinaten berechnen:

- Der Skalierungsfaktor ist das Verhältnis der Länge der Linie, die zur neuen *Grundlinie* wird, zur Länge der Original−*Grundlinie*
- Die Endpunkte von allen Linien lassen sich so berechnen: zuerst wird die Linienlänge mit dem Skalierungsfaktor multipliziert, und dann zu ihrem Winkel der Winkel der jeweiligen *Grundlinie* dazuaddiert. Danach werden diese (in Polarform relativ zum Anfangspunkt) vorliegenden Koordinaten in die kartesische Form gebracht:
  x−Koordinate = Länge der Linie * cos (Winkel)
  y−Koordinate = Länge der Linie * sin (Winkel)
  und schließlich zu den absoluten kartesischen Koordinaten des Anfangspunktes der Linie dazuaddiert.

Beispiel: Der Anfangspunkt der Original−*Grundlinie* ist in kartesischen Koordinaten angegeben; er ist gleichzeitig Anfangspunkt der Linie *Position_Linie1*. Daraus läßt sich der Endpunkt von *Position_Linie1* errechnen, der wiederum der Anfangspunkt von *Linie1* ist; wenn man den Endpunkt von *Linie1* berechnet, kann man sie zeichnen. Der Anfangspunkt von *Linie1* ist aber auch Anfangspunkt der neuen *Grundlinie*; da er jetzt ebenfalls in absoluten kartesischen Koordinaten vorliegt, lassen sich so alle weiteren darauf befindlichen Linien berechnen und zeichnen.

Programmdokumentation:

Das Programm liest zuerst die gewünschte maximale Rekursionstiefe (in die Variable *maximale_Rekursionstiefe*) ein, und schaltet dann in den Grafikmodus (VGA−Modus 12h: 640*480 Pixel bei 16 Farben; mit Hilfe des BGI−Treibers *egavga.bgi*). Danach werden den Variablen ihre Werte zugewiesen (Position der *Grundlinie*, Länge der Linien, Winkel etc.; s. oben) und die Funktion *zeichne_Fraktal* aufgerufen, die eine Linie als Parameter entgegennimmt und diese als *Grundlinie* benutzt; hier wird die Original−*Grundlinie* als Parameter übergeben.
Zur Speicherung der Koordinaten der Linien werden zwei Arten von Records benutzt: *TLinie* stellt eine ganze Linie dar; der Anfangspunkt ist in kartesischer Form in den Variablen *TLinie.x* und *TLinie.y* gespeichert, der Richtungsvektor der Linie ist durch *TLinie.Winkel* und *TLinie.Laenge* definiert (kartesische Koordinaten und Längen werden in Pixel gemessen, Winkel in Bogenmaß). *Grundlinie* ist eine Variable vom Typ *TLinie*. Der andere Record ist *TVektor*, der einen Vektor in Polarform darstellt; sein Winkel wird von *TVektor.Winkel* und seine Länge von *TVektor.Laenge* angegeben. Die beiden Variablen *Linie1* und *Linie2* sind vom Typ TVektor und geben die Richtungsvektoren der beiden kleinen Linien relativ zur (Original−)*Grundlinie* an. Ihre Anfangspunkte relativ zur *Grundlinie*, d.h. die Strecken vom Anfangspunkt der *Grundlinie* bis zu den Anfangspunkten der kleinen Linien, werden durch die Variablen *Position_Linie1* und *Position_Linie2* in Polarform gespeichert. Hier wird die Polarform deshalb verwendet, weil damit das addieren der Winkel und das Skalieren der Längen der Linien einfacher ist und

ohne Umrechnung geht.
Das eigentliche Zeichnen des Fraktals wird von der Funktion *zeichne_Fraktal* übernommen. Diese Funktion nimmt eine *Grundlinie* in der Variablen *neue_Grundlinie* (Variablentyp: *TLinie*) als Parameter entgegen und zeichnet auf diese *Grundlinie* die gesamte Figur und ruft sich danach rekursiv mit den beiden gerade gezeichneten kleinen Linien als neue *Grundlinie*n auf. Diese Funktion arbeitet folgendermaßen: zuerst wird die (als Parameter übergebene) *Grundlinie* gezeichnet. Danach wird die Variable *Rekursionstiefe*, die die Rekursionstiefe angibt, um 1 erhöht und überprüft, ob sie größer als die vom Benutzer eingegebene *maximale_Rekursionstiefe* ist. Wenn ja, wird sie wieder um 1 erniedrigt und die Funktion beendet. Dann wird der *Skalierungsfaktor* ausgerechnet (s.o.): das Verhältnis der Länge der übergebenen *Grundlinie* zur Länge der Original−*Grundlinie*; die Länge der neuen *Linie1* bzw. *Linie2* ist die Länge der Original−*Linie1* bzw. −*Linie2*, multipliziert mit dem *Skalierungsfaktor*. Dann werden beide neuen Linien gedreht: ihre Winkel berechnen sich aus der Summe des (absoluten) Winkels von *neue_Grundlinie* und den Winkeln von *Linie1* bzw. *Linie2* (s.o.). Dann werden die (absoluten) Koordinaten der Anfangspunkte von *neue_Linie1* bzw. *neue_Linie2* berechnet: die skalierte Länge von *Position_Linie1* bzw. *Position_Linie2* wird in kartesische Koordinaten umgewandelt (=> relative Position des Anfangspunkts von *Linie1* bzw. *Linie2*) und zu den Koordinaten des Anfangspunktes der *Grundlinie neue_Grundlinie* hinzuaddiert. Schließlich ruft sich die Funktion rekursiv mit *neue_Linie1* und *neue_Linie2* als *Grundlinie*n auf. Dann wird noch *Rekursionstiefe* um 1 erniedrigt und anschließend die Funktion beendet.

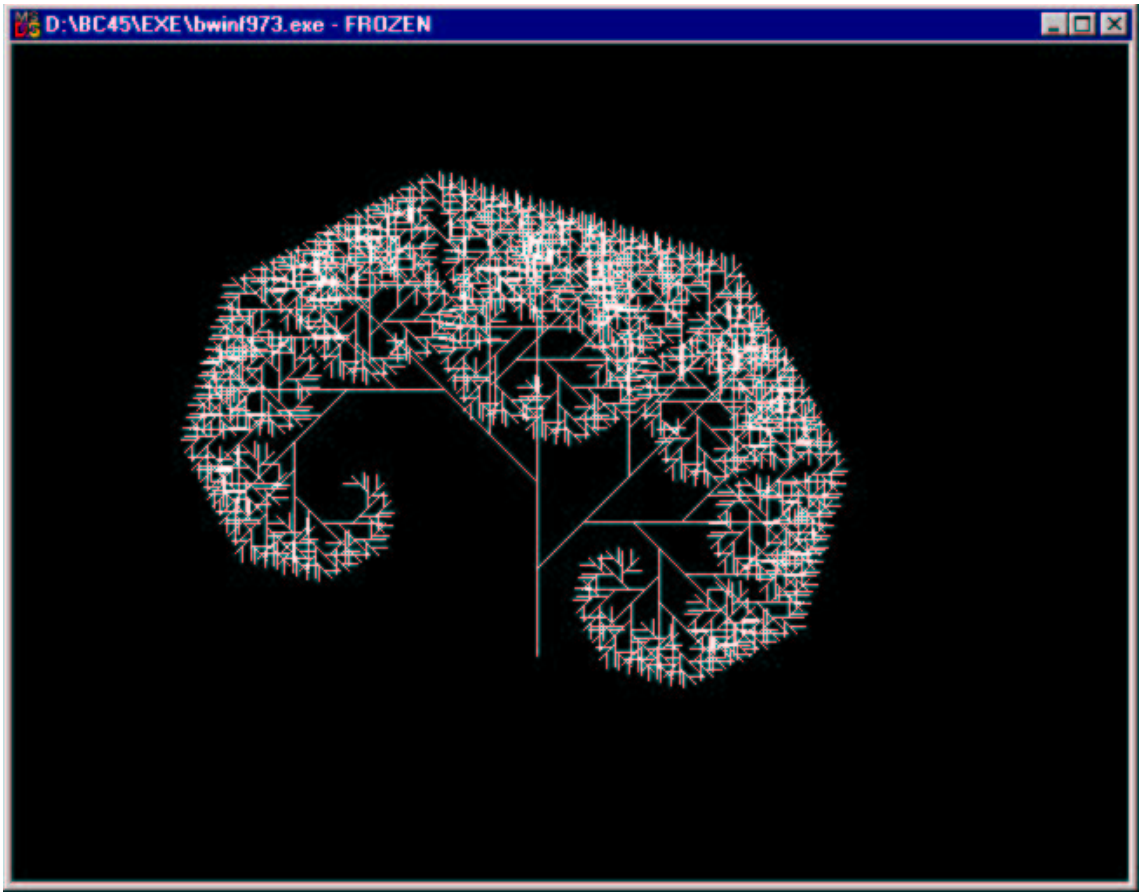
Programmcode:

```
#include <graphics.h> // line()
#include <conio.h>    // getch()
#include <stdio.h>    // printf()
#include <math.h>     // sin()

// wandelt Winkelangaben in Grad zu Bogenmaß um
#define Grad_zu_Rad(Winkel)   (Winkel*M_PI/180)

// rundet eine Gleitkommazahl (float) auf eine Ganzzahl (int), anstatt sie
// abzubrechen (der Borland C++ Run-Time-Library entnommen)
#define runde(Gleitkommazahl) floor(Gleitkommazahl+0.5)

typedef struct
        {
         float Winkel; // in Bogenmaß
         int Laenge;           // in Pixel
        } TVektor;             // ein Vektor in Polarform

typedef struct
        {
         int x;                // in Pixel
         int y;                // in Pixel
         float Winkel; // in Bogenmaß
         int Laenge;           // in Pixel
        } TLinie;              // eine Linie, deren Startvektor in kartesischer Form
                       // und deren Richtungsvektor in Polarform angegeben
        // ist

// die erste Grundlinie
TLinie Grundlinie;

// die beiden kurzen Linien oberhalb der Grundlinie:
// Linie1 und Linie2 sind ihre Richtungsvektoren (relativ zur Grundlinie);
// Position_Linie1 und Position_Linie2 sind die Startvektoren dieser beiden
// Linien (ebenfalls relativ zum Startvektor der Grundlinie); sie sind in
// Polarform angegeben, damit sie später leichter gedreht werden können
TVektor Linie1,Linie2,Position_Linie1,Position_Linie2;

int Rekursionstiefe=0;               // speichert die aktuelle Rekursionstiefe
int maximale_Rekursionstiefe; // die vom Benutzer eingegebene maximale
                       // Rekursionstiefe

void zeichne_Linie(TLinie Linie)
{
 // die Koordinaten des Endpunktes der zu zeichnenden Linie in kartesischer
 // Form und absolut (d.h. relativ zur linken oberen Ecke des Bildschirms)
 int Ende_x,Ende_y;

 Ende_x=Linie.x+runde(Linie.Laenge*cos(Linie.Winkel));
 Ende_y=Linie.y+runde(Linie.Laenge*sin(Linie.Winkel));

 line(Linie.x,Linie.y,Ende_x,Ende_y);
}

void zeichne_Fraktal(TLinie neue_Grundlinie)
{
 TLinie neue_Linie1,neue_Linie2;     // für den nächsten rekursiven Aufruf
 float Skalierungsfaktor;

 // zeichne aktuelle Grundlinie
 zeichne_Linie(neue_Grundlinie);

 // Wenn die maximale Rekursionstiefe überschritten wird, dann nicht
```

```
 // weitermachen, sondern eine Ebene zurückspringen
 if(++Rekursionstiefe > maximale_Rekursionstiefe)
 {
  Rekursionstiefe--;
  return;
 }

 // alle Linien werden im selben Verhältnis verkleinert, wie die Grundlinie
 // verkleinert worden ist
 Skalierungsfaktor=(float)neue_Grundlinie.Laenge/Grundlinie.Laenge;

 // skaliere die Linienlängen
 neue_Linie1.Laenge=Skalierungsfaktor*Linie1.Laenge;
 neue_Linie2.Laenge=Skalierungsfaktor*Linie2.Laenge;

 // drehe Linien
 neue_Linie1.Winkel=neue_Grundlinie.Winkel+Linie1.Winkel;
 neue_Linie2.Winkel=neue_Grundlinie.Winkel+Linie2.Winkel;

 // berechne Anfangsposition der neuen Linien
 neue_Linie1.x=neue_Grundlinie.x+runde((Position_Linie1.Laenge
       *Skalierungsfaktor)*cos(neue_Grundlinie.Winkel+Position_Linie1.Winkel));
 neue_Linie1.y=neue_Grundlinie.y+runde((Position_Linie1.Laenge
       *Skalierungsfaktor)*sin(neue_Grundlinie.Winkel+Position_Linie1.Winkel));
 neue_Linie2.x=neue_Grundlinie.x+runde((Position_Linie2.Laenge
       *Skalierungsfaktor)*cos(neue_Grundlinie.Winkel+Position_Linie2.Winkel));
 neue_Linie2.y=neue_Grundlinie.y+runde((Position_Linie2.Laenge
       *Skalierungsfaktor)*sin(neue_Grundlinie.Winkel+Position_Linie2.Winkel));

 // gehe eine Rekursionsebene tiefer
 zeichne_Fraktal(neue_Linie1);
 zeichne_Fraktal(neue_Linie2);

 // O.K., alles erledigt, nun eine Ebene zurück und da weitermachen
 Rekursionstiefe--;
}

// Hauptprogramm
int main()
{
 // ich gehe vom VGA-Grafikmodus 640 mal 480 Pixel aus (Modus 12h)
 int Grafiktreiber=VGA,Grafikmodus=VGAHI;

 printf("Maximale Rekursionstiefe ? ");
 scanf("%d",&maximale_Rekursionstiefe);

 initgraph(&Grafiktreiber,&Grafikmodus,"");

 // falls es Grafikprobleme gibt
 if(graphresult()!=grOk)
 {
  printf("Fehler! Beende Programm! (BGI-Fehler %d)\n",graphresult);
  return 1;
 }

 Grundlinie.x=200;
 Grundlinie.y=400;
 Grundlinie.Winkel=Grad_zu_Rad(0);
 Grundlinie.Laenge=300;

 Position_Linie1.Winkel=Grad_zu_Rad(-160);
 Position_Linie1.Laenge=75;
 Linie1.Winkel=Grad_zu_Rad(-45);
 Linie1.Laenge=250;

 Position_Linie2.Winkel=Grad_zu_Rad(-20);
 Position_Linie2.Laenge=300;
 Linie2.Winkel=Grad_zu_Rad(-80);
 Linie2.Laenge=250;

 // Originalfigur ist bekannt => mach draus ein Fraktal !
 zeichne_Fraktal(Grundlinie);

 // warten, bis der User das Bild fertig angeschaut hat und eine Taste drückt
 getch();

 // Grafikmodus aus !
 closegraph();

 return 0;
}
```

Programmablaufprotokoll (Screenshots):

Lösungen zu Aufgabe 3: Bilder mit veränderter Ausgangslage der Borsten
Etwas asymmetrisch: 10 Rekursionsstufen von (Ausschnitt aus dem Programmlisting; der geänderte Code):

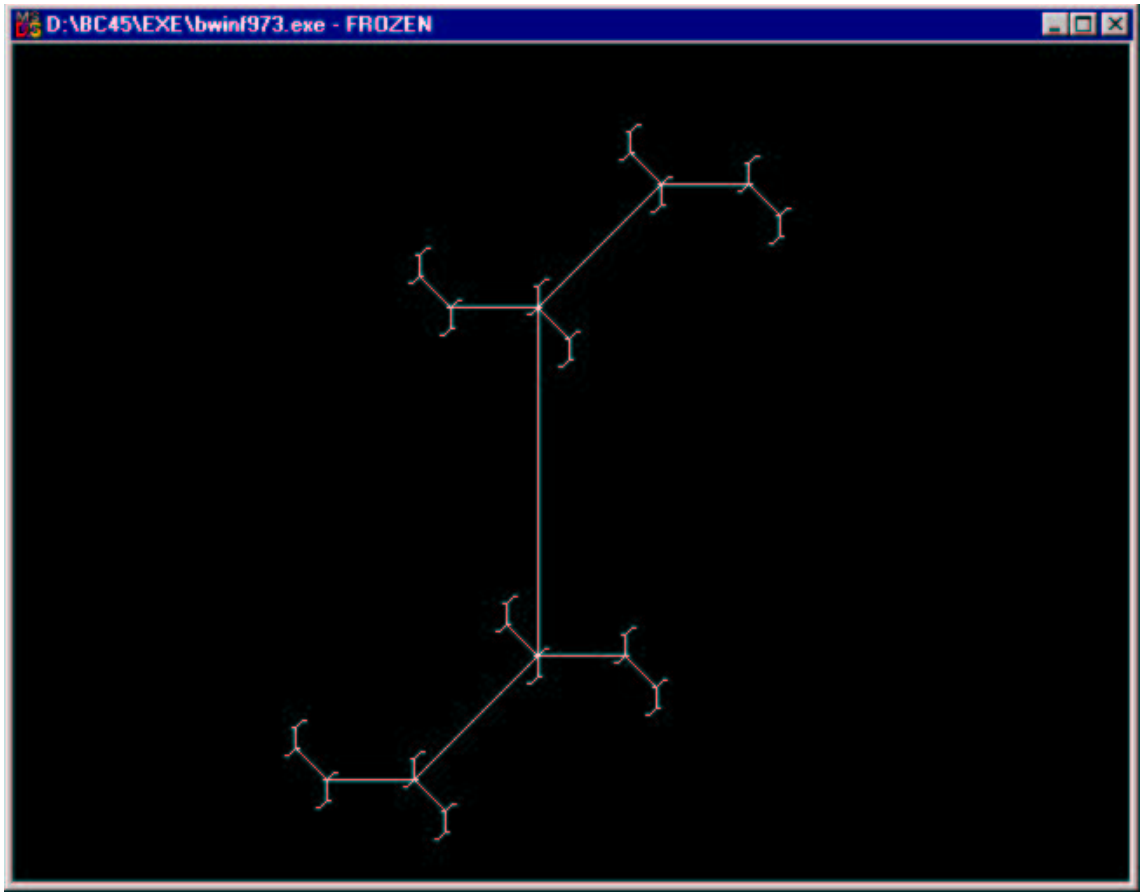


Grundlinie.x=300;
Grundlinie.y=350;
Grundlinie.Winkel=Grad_zu_Rad(−90);
Grundlinie.Laenge=200;

Position_Linie1.Winkel=Grad_zu_Rad(0);
Position_Linie1.Laenge=100;
Linie1.Winkel=Grad_zu_Rad(−45);
Linie1.Laenge=150;

Position_Linie2.Winkel=Grad_zu_Rad(0);
Position_Linie2.Laenge=50;
Linie2.Winkel=Grad_zu_Rad(+45);
Linie2.Laenge=150;

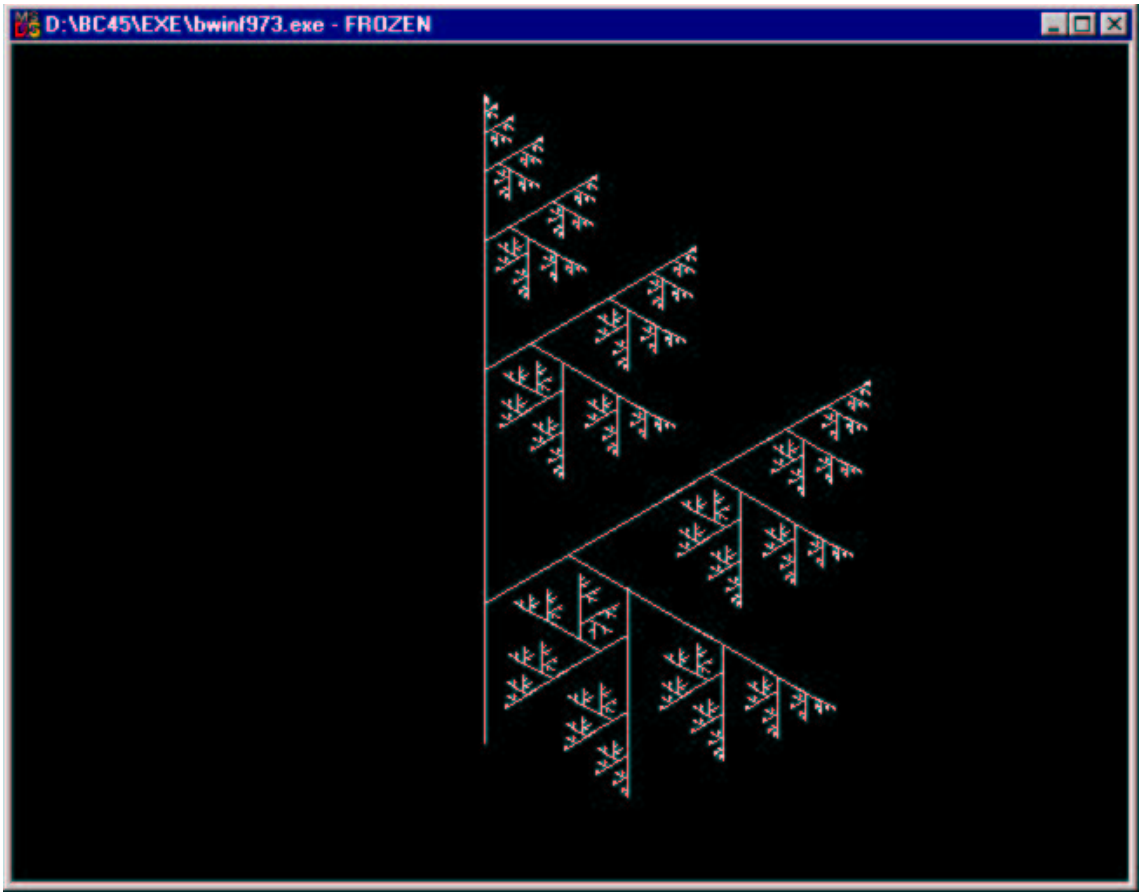
Probiert mal 20 Rekursionsstufe aus! Da pro *Grundlinie* immer 2 kurze Linien gezeichnet werden müssen und pro Rekursionsebene 2 *Grundlinie*n entstehen, ergibt sich, dass insgesamt $2^{\text{Rekursionsebene}}$ Linien pro Ebene gezeichnet werden müssen! Da kann man auch dem schnellsten Computer beim Zeichnen richtig zuschauen!



Grundlinie.x=300;
Grundlinie.y=350;
Grundlinie.Winkel=Grad_zu_Rad(−90);
Grundlinie.Laenge=200;

Position_Linie1.Winkel=Grad_zu_Rad(0);
Position_Linie1.Laenge=0;
Linie1.Winkel=Grad_zu_Rad(−135);
Linie1.Laenge=100;

Position_Linie2.Winkel=Grad_zu_Rad(0);
Position_Linie2.Laenge=200;
Linie2.Winkel=Grad_zu_Rad(+45);
Linie2.Laenge=100;



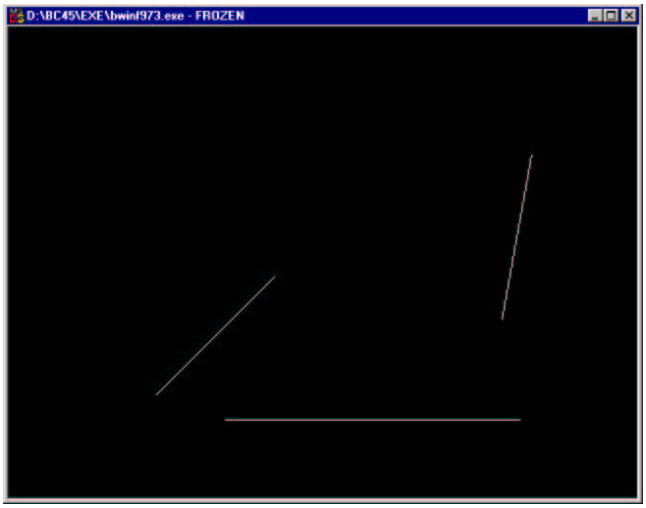
Ich kann auch Blätter machen... (10 Rekursionsstufen)

```
Grundlinie.x=270;
Grundlinie.y=400;
Grundlinie.Winkel=Grad_zu_Rad(−90);
Grundlinie.Laenge=200;

Position_Linie1.Winkel=Grad_zu_Rad(0);
Position_Linie1.Laenge=170;
Linie1.Winkel=Grad_zu_Rad(0);
Linie1.Laenge=110;

Position_Linie2.Winkel=Grad_zu_Rad(0);
Position_Linie2.Laenge=80;
Linie2.Winkel=Grad_zu_Rad(+60);
Linie2.Laenge=140;
```

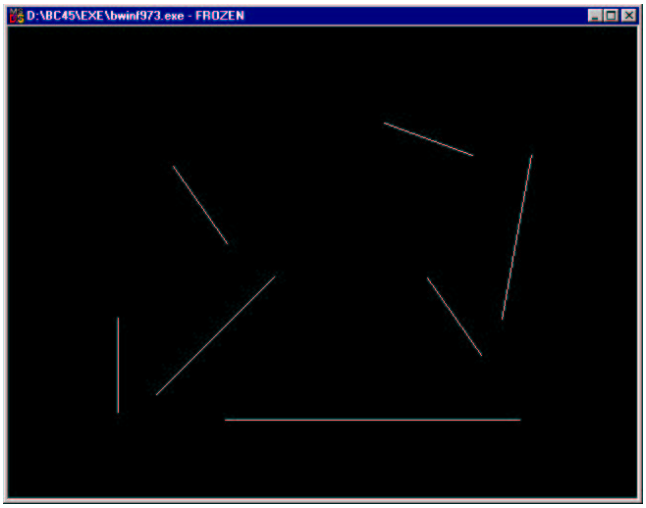
Anmerkungen:

– warum so viele Screenshots ???

Lösungen zur Aufgabe 2: Bilder der ersten 6 Stufen der angegebenen Borstenanordnung



mit Rekusionstiefe 1

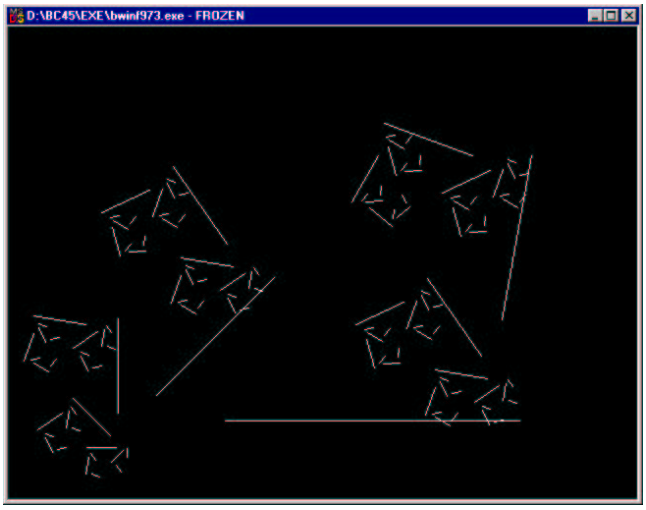


mit Rekursionstiefe 2

mit Rekursionstiefe 3

mit Rekursionstiefe 4

mit Rekursionstiefe 5



mit Rekursionstiefe 6

Aufgabe 4: Wetter in Quadratien

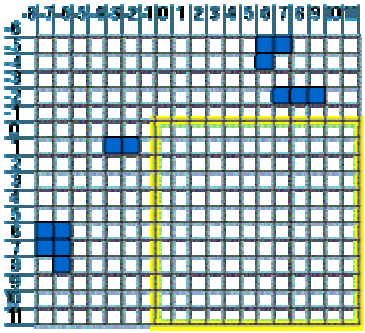
Aufgabenstellung:

Quadratien ist ein quadratisches Gebiet aus quadratischen Feldern. Das Feld in der Nordwest−Ecke hat die Zeilennummer 0 und die Spaltennummer 0.

Das Wetter in Quadratien wird durch quadratische Wolken bestimmt, die genau ein Feld groß sind. Solche Wolken rücken getaktet über Quadratien vor, und zwar von Norden nach Süden 2 Felder pro Takt und, in einer anderen Höhe, von Westen nach Osten 3 Felder pro Takt. Es regnet überall dort, wo sich nach einem Vorrücken sowohl eine Nord−Süd− als auch eine West−Ost−Wolke befindet. Wolken, aus denen es regnet, lösen sich auf.

Die Wolkenvorhersage gibt an, an welchen Stellen (in der Form: Zeilennummer, Spaltennummer) sich zum aktuellen Zeitpunkt Wolken befinden. Daraus läßt sich dann ermitteln, wo es in Quadratien regnen wird, denn es werden nur solche Wolken angegeben, die über Quadratien hinwegziehen werden.

Beispiel:
Wolkenvorhersage:
−5/6 −4/6 −2/7 7/−6 −2/9 −5/7 1/−2 8/−6
6/−6 7/−7 6/−7 1/−3 −2/8.



Es wird jeweils einmal regnen an den Stellen 1,6, 1,7 und 8,9.

Aufgabe:
Schreibe ein Programm, welches folgendes leistet:
1. Einlesen der Größe von Quadratien (Anzahl Zeilen bzw. Spalten)
2. Einlesen einer Wolkenvorhersage
3. Ausgabe, wo in Quadratien wie oft Regen fällt

Sende uns 3 Beispiele, darunter eines für ein Quadratien der Größe 10 x 10 und folgender Wolkenvorhersage:
2/−3 −5/5 −3/4 1/−4 6/−12 −3/5 −7/5 3/−10
−6/6 6/−11 −7/4 3/−4 −4/5 −3/3 −6/9 2/−4.

Lösung:

Lösungsidee:

Am einfachsten und anschaulichsten ist eine Simulation der Bewegung: In einer Endlosschleife werden alle Wolken mit ihrer spezifischen Geschwindigkeit bewegt und nach jedem Takt untersucht, ob es Wolken gibt, die sich an der gleichen Position innerhalb Quadratiens befinden; wenn ja, werden diese als „abgeregnet“ markiert und und die Position des Regens in einer Liste gespeichert. Die Schleife wird abgebrochen, wenn sich keine Wolke mehr im Land befindet.

Programmdokumentation:

Die Variable *Laenge* gibt die Seitenlänge des quadratischen Quadratiens an, in der Variablen *Anzahl_Wolken* wird die Anzahl aller Wolken abgespeichert.
Die Koordinaten jeder Wolke werden zuerst in die Variable *Wolke_temporaer* eingelesen und dann in eines von zwei Arrays einsortiert: das Array *Wolke_West* speichert die Koordinaten der Wolken, die aus Westen kommen, *Wolke_Nord* die Koordinaten der Wolken aus Norden. Während des Einsortierens wird auch die Anzahl der westlichen und nördlichen Wolken gespeichert, dies geschieht in *Anzahl_Wolken_West* und *Anzahl_Wolken_Nord* (damit bei der Simulation nicht die ganzen Arrays untersucht werden müssen, sondern nur derjenige Teil, der tatsächlich von Wolkenkoordinaten belegt ist). Das Einsortieren funktioniert mit Hilfe der Wolkenkoordinaten: z.B. von einer Wolke, deren Spaltenkoordinate kleiner als 0 ist, d.h. sie befindet sich westlich des Landes, wird angenommen, dass sie es nach Osten überquert. Danach folgt die eigentliche Simulation: eine Bedingungsschleife wird dazu verwendet, wobei jeder Schleifendurchlauf einen Takt darstellt. Mit jedem Takt werden die Wolken zuerst bewegt, d.h. die Spaltenkoordinate jeder Wolke aus Westen wird um 3 und die Zeilenkoordinate jeder Wolke aus Norden wird um 2 erhöht. Danach wird die Position jeder Wolke aus Norden mit allen Wolken aus Westen verglichen. Wenn beide Koordinaten übereinstimmen, dann regnet es dort. Das Array *Regen* speichert dabei die Koordinaten dieser Orte sowie die Anzahl der Regenschauer, falls sich der Regenschauer noch innerhalb des Landes befand. Zuerst werden deshalb alle Einträge dieses Arrays nach dem aktuellen Ort durchsucht. Wenn er gefunden wurde, wird einfach die Anzahl der dortigen Schauer um 1 erhöht, sonst ein neuer Eintrag mit dem aktuellen Ort angelegt (indem im Array *Regen* hinter den anderen Einträgen die Koordinaten des neuen Regenorts eingetragen werden, die Anzahl der Schauer an diesem Ort auf 1 gesetzt wird und die Variable *Anzahl_Regen* um 1 erhöht wird). Danach können die beiden Wolken gelöscht werden; dem ’Löschen’ einer Wolke entspricht dabei, dass ihre Zeilen− bzw. Spaltenkoordinaten einen bestimmten, reservierten Wert zugewiesen bekommen (Konstante *ABGEREGNET*). Um die jeweilige letzte verwendete Position in diesem Array zur Verfügung zu haben, gibt es *Anzahl_Regen*, die die aktuelle Anzahl der Orte mit Regen speichert. Schließlich gibt es noch *flag_aktive_Wolken*: eine bool’sche Variable, die angibt, ob es noch Wolken in Quadratien gibt; dazu wird sie bei jedem Schleifendurchgang auf 0 gesetzt und, falls beim Vergleich der Wolkenkoordinaten mit den Landesgrenzen festgestellt wird, dass sich noch Wolken im Land befinden, wieder auf 1 gesetzt. Wenn diese Variable am Ende der Schleife 0 geblieben ist, kann die Schleife beendet werden und Bilanz gezogen werden.

Programmablauf als halbformale Programmbeschreibung:

Einlesen der Landesgröße (=Seitenlänge), Kontrolle der Eingabe auf unsinnige Werte, speichern der Landesgröße in der Variablen *Laenge*
Einlesen der Anzahl der Wolken, Kontrolle der Eingabe, speichern in der Variablen *Anzahl_Wolken*

in einer Zählschleife:

> von jeder Wolke die Koordinaten in die Variable *Wolke_temporaer* einlesen
> wenn die Wolke nördlich von Quadratien ist, ihre Koordinaten dem Array *Wolke_Nord* hinzufügen
> wenn die Wolke westlich von Quadratien ist, ihre Koordinaten dem Array *Wolke_West* hinzufügen

Schleifenende

in einer Bedingungsschleife:

> in Zählschleifen alle Wolken um 1 Takt bewegen (Zeilen− bzw. Spaltenkoordinaten entspr. erhöhen)
> die Koordinaten von jeder Wolke aus Norden mit jeder Wolke aus Westen vergleichen (mit zwei ineinanderliegenden Zählschleifen)
> wenn sich bei Übereinstimmung der Ort innerhalb der Landes befindet, Position im Array *Regen* speichern, denn dort regnet es, und beide Wolken als abgeregnet markieren

Ende der Bedingungsschleife − wiederholen, bis keine Wolke mehr im Land ist

Meldung ausgeben, wo es in Quadratien wie oft regnet

Programmcode:

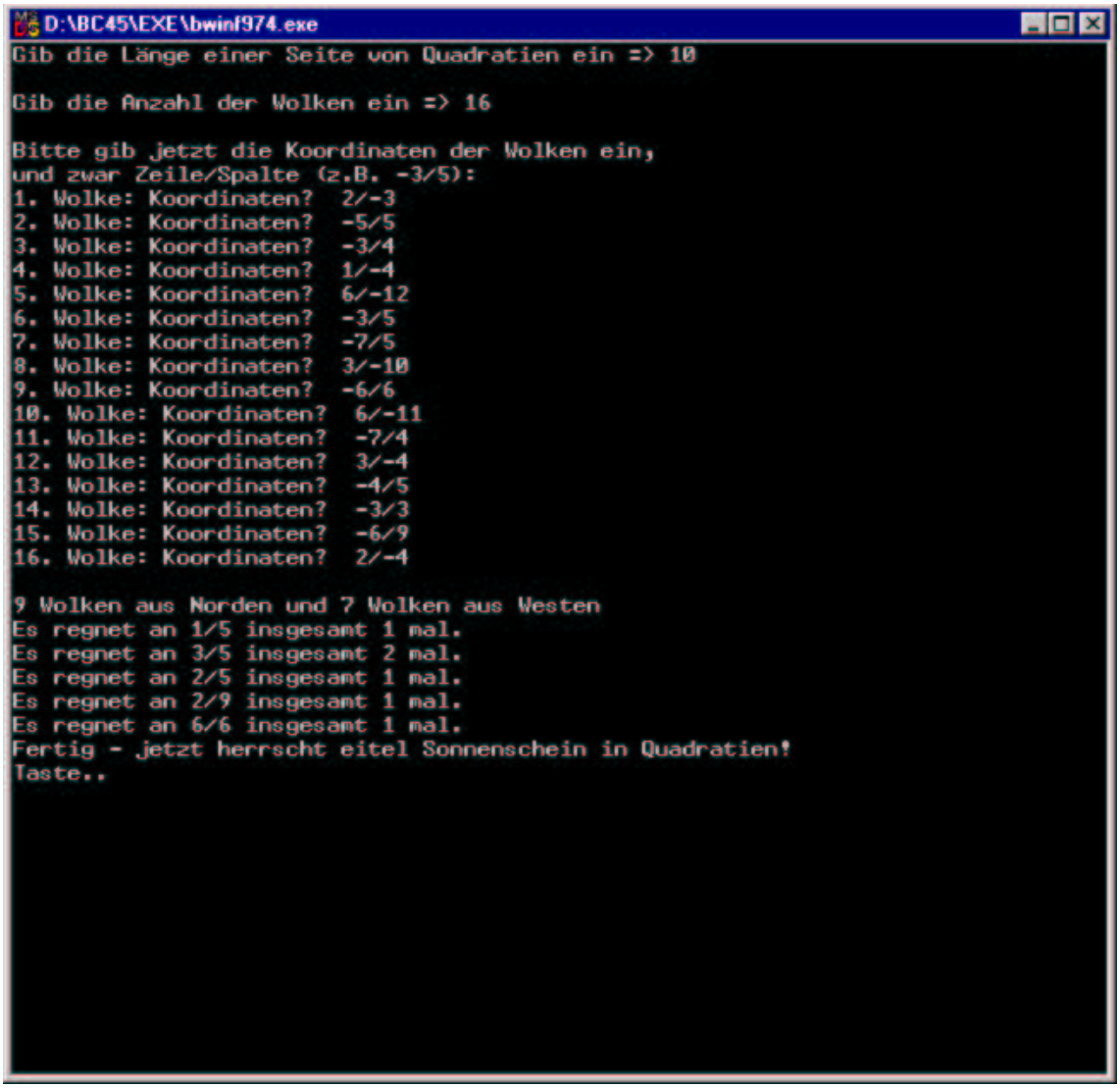
```
#include <stdio.h>    // printf()
#include <conio.h>    // getch()
#include <limits.h>   // INT_MAX

#define MAX_WOLKEN    50
#define ABGEREGNET    INT_MAX

typedef struct
       {
        int Zeile;
        int Spalte;
       } TWolke;      // das ist eine Wolke (Position)

typedef struct
       {
        int Zeile;
        int Spalte;
        int Anzahl;
       } TRegen;      // das ist ein Ort, an dem es regnet

int main()
{
 int Laenge=0;        // Seitenlänge von Quadratien
 int Anzahl_Wolken=0;         // Gesamtanzahl der vorhandenen Wolken
 int Anzahl_Wolken_West=0;    // Anzahl der Wolken aus Westen
 int Anzahl_Wolken_Nord=0;    // Anzahl der Wolken aus Norden
 int Anzahl_Regen=0;          // Anzahl der Orte, an denen es regnet
 TWolke Wolke_West[MAX_WOLKEN];      // enthält die Koordinaten der westlichen Wolken
 TWolke Wolke_Nord[MAX_WOLKEN];      // enthält die Koordinaten der nördlichen Wolken
 TWolke Wolke_temporaer;      // zum Einlesen der Koordinaten

 // es kann höchstens halb so oft regnen, wie Wolken da sind
 TRegen Regen[MAX_WOLKEN/2];

 int i,j,k;                   // Zählvariablen
 char eingabe[10];            // Eingabestring

 // gibt an, ob sich mindestens eine nicht-abgeregnete Wolke im Land befindet
 int flag_aktive_Wolken;

 // Abfrage der Größe des Landes
 printf("Gib die Länge einer Seite von Quadratien ein => ");
 scanf("%d",&Laenge);
 printf("\n");

 // Abfrage der Anzahl der Wolken
 printf("Gib die Anzahl der Wolken ein => ");
 scanf("%d",&Anzahl_Wolken);
 printf("\n");

 // Einlesen der Koordinaten der Wolken
 printf("Bitte gib jetzt die Koordinaten der Wolken ein,\nund zwar Zeile/Spalte"
        " (z.B. -3/5):\n");
 gets(eingabe);

 for(i=0;i<Anzahl_Wolken;i++)
 {
  printf("%d. Wolke: Koordinaten?  ",i+1);
  gets(eingabe);

  // bei einem Eingabefehler: Eingabe wiederholen lassen
  if(sscanf(eingabe,"%d/%d",&Wolke_temporaer.Zeile,&Wolke_temporaer.Spalte)!=2)
  {
   printf("Fehler! Nochmal:\n");
   i--;
   continue;
  }

  // Ist die Wolke nördlich von Quadratien?
  if(Wolke_temporaer.Zeile<0)
  {
   // ja, diese Wolke kommt von Norden -> einsortieren
   Wolke_Nord[Anzahl_Wolken_Nord].Zeile=Wolke_temporaer.Zeile;
   Wolke_Nord[Anzahl_Wolken_Nord].Spalte=Wolke_temporaer.Spalte;
   Anzahl_Wolken_Nord++;      // 1 Wolke mehr aus nördl. Richtung registriert
  }
  // Ist die Wolke nördlich von Quadratien?
  else if(Wolke_temporaer.Spalte<0)
  {
   // ja, diese Wolke kommt aus Westen -> einsortieren
   Wolke_West[Anzahl_Wolken_West].Zeile=Wolke_temporaer.Zeile;
   Wolke_West[Anzahl_Wolken_West].Spalte=Wolke_temporaer.Spalte;
   Anzahl_Wolken_West++;      // 1 Wolke mehr aus westl. Richtung registriert
  }
 }
```

```
printf("\n%d Wolken aus Norden und %d Wolken aus Westen\n",
       Anzahl_Wolken_Nord,Anzahl_Wolken_West);


for(i=0;i<Anzahl_Wolken/2;i++) Regen[i].Anzahl=0;  // Werte auf 0 setzen

do
{
 flag_aktive_Wolken=0;                      // noch gibt's keine aktive Wolken

 for(i=0;i<Anzahl_Wolken_Nord;i++)  // mit allen nördlichen Wolken...
 {
  // ...um 2 nach Süden, falls sie noch nicht abgeregnet ist
  if(Wolke_Nord[i].Zeile!=ABGEREGNET) Wolke_Nord[i].Zeile+=2;

  // befindet sich die Wolke noch in Quadratien?
  // wenn ja, dann ist sie also noch 'aktiv'
  if(Wolke_Nord[i].Zeile<Laenge) flag_aktive_Wolken=1;
 }

 for(i=0;i<Anzahl_Wolken_West;i++)
 {
  // alle westl. Wolken um 3 nach Osten, falls sie noch nicht abgeregnet sind
  if(Wolke_West[i].Spalte!=ABGEREGNET) Wolke_West[i].Spalte+=3;

  // befindet sich die Wolke noch in Quadratien?
  // wenn ja, dann ist sie also noch 'aktiv'
  if(Wolke_West[i].Spalte<Laenge) flag_aktive_Wolken=1;
 }

 // Wo regnet es? Alle Kombinationen von westl./nördl. Wolken untersuchen!
 for(i=0;i<Anzahl_Wolken_Nord;i++)
 {
  for(j=0;j<Anzahl_Wolken_West;j++)
  {

   // wenn zwei Wolken auf einem Quadrat sind und sich im Land befinden...
   // (eigentlich können sich keine Wolken westlich oder nördlich von
   // Quadratien treffen – ich habe die Überprüfung aber der Vollständigkeit
   // wegen aufgenommen; (Wolke_Nord[i].Spalte>=0) und
   // (Wolke_Nord[i].Zeile>=0) sind hier unnötig, da der Benutzer keine Wolken
   // eingeben darf, die nordwestlich von Quadratien sind)
   if((Wolke_Nord[i].Zeile==Wolke_West[j].Zeile)
      &&(Wolke_Nord[i].Spalte==Wolke_West[j].Spalte)
      &&(Wolke_Nord[i].Spalte<Laenge)&&(Wolke_Nord[i].Spalte>=0)
      &&(Wolke_Nord[i].Zeile<Laenge)&&(Wolke_Nord[i].Zeile>=0))
   {
    // alle Regenorte durchschauen, ob es an diesem Ort schon mal geregnet hat
    for(k=0;k<Anzahl_Regen;k++)
    {
     // wenn der Ort schon in der Regenliste vorhanden ist...
     if((Regen[k].Zeile==Wolke_Nord[i].Zeile)
       &&(Regen[k].Spalte==Wolke_Nord[i].Spalte))
     {
       Regen[k].Anzahl++;    // ...eintragen, dass es dort einmal mehr regnet
       break;         // fertig – vorzeitig raus aus der Suchschleife
     }
    }

    // die Suchschleife wurde komplett durchlaufen => Eintrag in der Regenliste
    // noch nicht vorhanden => neuen erzeugen
    if(k==Anzahl_Regen)
    {
     // Koordinaten des Quadrats in der Regenliste registrieren
     Regen[Anzahl_Regen].Zeile=Wolke_Nord[i].Zeile;
     Regen[Anzahl_Regen].Spalte=Wolke_Nord[i].Spalte;

     // ein Regenquadrat mehr, dort hat es einmal geregnet
     Regen[Anzahl_Regen].Anzahl=1;
     Anzahl_Regen++;
    }

    // beide Wolken als abgeregnet markieren (es reicht aus, nur jeweils
    // eine der Koordinaten zu markieren)
    Wolke_Nord[i].Zeile=ABGEREGNET;
    Wolke_West[j].Spalte=ABGEREGNET;
    continue;
   }
  }
 }

// solange es aktive Wolken gibt, also Wolken im Land sind,
// die nicht abgeregnet sind
}while(flag_aktive_Wolken==1);

// Ergebnis ausgeben
for(i=0;i<Anzahl_Regen;i++) printf("Es regnet an %d/%d insgesamt %d mal.\n",
       Regen[i].Zeile,Regen[i].Spalte,Regen[i].Anzahl);

printf("Fertig – jetzt herrscht eitel Sonnenschein in Quadratien!\nTaste..\n");

getch();
```

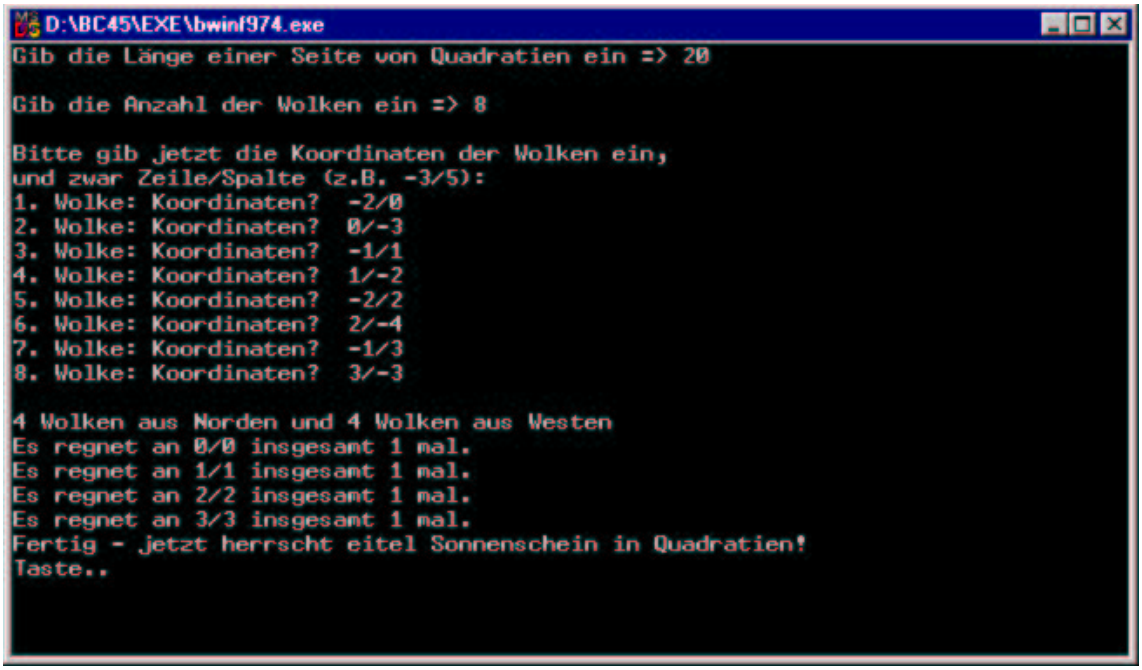
```
 return 0;
}
```

Programmablaufprotokoll (Screenshots):

Besonderheit: hier regnet es auch ausserhalb von Quadratien an der Position 8/9



Lösung der Aufgabe



Schöne Reihenfolge, nicht?

Anmerkungen:

- die maximale Anzahl an Wolken ist durch die Konstante *MAX_WOLKEN* auf 50 festgelegt; kann je nach Bedarf erhöht werden
- das Programm macht keinerlei Prüfungen auf falsche Eingaben, z.B. negative Länge von Quadratien o.ä.; der Benutzer muss selbst auf die Eingabe von richtigen und sinnvollen Werten achten
- der Benutzer muss vor der Eingabe der Wolkenkoordinaten die Anzahl der Wolken eingeben; in der Aufgabenstellung steht ja nicht ausdrücklich drin, dass der Benutzer die Wolkenkoordinaten, ohne vorher ihre Anzahl festzustellen, eingeben können muss
- wenn mehrere Wolken mit den selben Koordinaten eingegeben werden, die Wolken also übereinander liegen, dann regnet sich nur immer eine dieser Wolken ab; die anderen ziehen normal weiter

Aufgabe 5: Nach der Party

Aufgabenstellung:

Das Fest bei Katrin Käfer war ein großer Erfolg. Nur eine Kleinigkeit trübt ihre Hochstimmung: Der von Katrin heimlich verehrte Robert hatte ihre Einladung zum Fest mit der Bemerkung abgelehnt, daß er keine Lust habe, seine Zeit auf so einer langweiligen Veranstaltung zu verschwenden. Mit der Verehrung ist es nach dieser Bemerkung natürlich vorbei, aber Katrin hofft nun, daß Robert von möglichst vielen verschiedenen Personen hört, wie superspitze das Fest bei ihr war, und sich entsprechend ärgert.
Wie oft Robert von dem Fest hört, läßt sich ausrechnen, wenn man weiß, wer alles auf dem Fest war, welche Personen im weitläufigen Bekanntenkreis von Katrin die Festgäste kennen, welche dieser Personen sich untereinander kennen und wer alles Robert kennt.
Sie vermutet ganz richtig, daß jeder, der etwas von dem Fest hört, dies auch weitererzählen wird. Andererseits wird aber niemand mehrfach der gleichen Person von dem Fest erzählen, auch wenn der Betreffende von mehreren verschiedenen Personen davon erzählt bekommt. Natürlich kennen sich alle Festgäste untereinander. Und natürlich wird sie selbst, obwohl sie Robert ja kennt, ihm niemals auch nur ein einziges Wort von ihrem Fest berichten. Auch ihren sonstigen nicht−eingeladenen Bekannten wird sie nicht von ihrem Fest vorschwärmen.

Beispiel:
Auf dem Fest waren Freddy, Ina und Vera.
Freddy kennt Robert und Lothar.
Ina kennt Lothar und Hans.
Vera kennt Robert und Hans.
Robert kennt Lothar und Volker.
Volker kennt Kirsten.
Kirsten kennt Robert.
Hans kennt Lothar und Robert.

(Wenn Robert Lothar kennt, dann kennt auch Lothar Robert, auch wenn dies nicht explizit angegeben wurde.)
In diesem Fall hört Robert viermal von dem Fest, nämlich von Freddy, Hans, Lothar und Vera.

Aufgabe:
Schreibe ein Programm, das nach Eingabe der Gäste und der Bekanntschaften im Bekanntenkreis von Katrin ermittelt, wie oft Robert von dem Fest hört. Die Eingabe der Bekanntschaften muß dabei nicht unbedingt in der Form zweier Namen erfolgen, sondern geeignete Kürzel sind ebenfalls möglich.
Sende uns drei Beispiele von Programmläufen, darunter eines mit folgenden Festgästen und Bekanntschaften:

Auf dem Fest waren Elisabeth, Christoph, Jochen und Gaby.
Gaby kennt Muriel, Helga, Cornelia und Bettina.
Bettina kennt Reinhard, Helga und Cornelia.
Elisabeth kennt Peter, Michael, Muriel und Herbert.
Robert kennt Ulrich, Herbert und Cornelia.
Jochen kennt Herbert, Janine und Robert.
Ulrich kennt Werner und Wolfgang.
Wolfgang kennt Robert und Werner.
Werner kennt Robert und Andrea.
Andrea kennt Robert und Wolfgang.
Janine kennt Reinhard und Robert.
Christoph kennt Peter, Michael und Herbert.
Peter kennt Robert und Herbert.
Muriel kennt Janine, Reinhard und Robert.
Michael kennt Robert.

Lösung:

Lösungsidee:

Entscheidend ist, wie man die Personen als Datenstruktur darstellt. Von allen Personen muss bekannt sein, wie sie heißen sowie wen sie kennen. Dabei kann man alle Personen gleich behandeln: Partygäste sind nicht anders als deren Bekannte; sie wissen lediglich von Anfang an von der Party und kennen sich untereinander. Und Robert ist auch 'nur' einer der Bekannten. Das naheliegendste, wie man die Bekanntschaftsverhältnisse auf dem Papier skizzieren kann, ist durch Doppelpfeile zwischen den Personen. Auf dem Computer sind Zeigervariablen (=Pointer) die Entsprechung zu den Pfeilen. Die Personen werden durch Records dargestellt, die alle für die Personen relevanten Daten speichern. Es hat also jede Person in ihrem Record mehrere Pointer, die auf die Records von Bekannten zeigen können. Angenommen, Person 1 kennt Person 2, dann bedeutet das, dass Person 1 einen Pointer auf Person 2 besitzt; da ja Person 2 natürlich dann auch Person 1 kennt, hat ebenso Person 2 einen Pointer auf Person 1. Daraus ergibt sich ein binärer Baum (mit Querverbindungen zwischen den Ästen untereinander), der sich am Besten mit Rekursion analysieren läßt. Man muss nur durch Rekursion alle Wege von einem der Partygäste bis zu Robert verfolgen und dann auswerten, wie viele der direkten Bekannten von Robert von der Party erfahren haben. Es reicht aus, die Rekursion bei einem einzigen der Partygäste zu starten, weil sich die Gäste sowieso alle untereinander kennen, man muss im Programm also gar nicht implementieren, dass die anderen Partygäste von der Party wissen! Zu beachten ist aber auch, dass man bei Querverbindungen in binären Baum die Rekursion stoppen muss, damit sie nicht „im Kreis" läuft (z.B. bei der Situation: Peter kennt Veronika, diese kennt Max, und Max kennt Peter darf es nicht vorkommen, dass Peter Veronika von der Party erzählt, Veronika es an Max weiter erzählt und dieser es schließlich wieder an Peter weitererzählt, der Veronika davon erzählt...). Also muss jeder Rekursionspfad markiert werden, damit die Rekursion stoppen kann, wenn sie auf einen bereits gegangenen Pfad trifft.

Programmdokumentation:

Der Record, der eine Person (*TPerson*) darstellt, beinhaltet folgende Variablen:

- *Name*: ein String, der den Namen der Person enthält;
- *von_Party_gehoert*: eine bool'sche Variable, die angibt, ob die Person schon von der Party gehört hat (=1);
- *Anzahl_Bekanntschaften*: gibt an, wie viele andere Leute diese Person kennt;
- *Bekannter*: ein Array von Pointern (vom Typ *TPerson*), von denen jeder auf den Record einer anderen Person zeigen kann; alle Personen im Bekanntenkreis werden durch je einen solchen Record im Array *Person* dargestellt.

Zuerst werden in einer Endlosschleife die Namen der Partygäste eingelesen und zu jedem Namen gleich ein Record im Array *Person* angelegt (dies geschieht mit der Funktion *registriere_neue_Person*, die den Namen in einen neuen Record in *Person* einträgt und die anderen Werte auf 0 setzt) und die Bekanntschaft mit allen bisher eingegebenen Partygästen (in beide Richtungen) registriert (durch die Funktion *registriere_Bekanntschaft*, die die beiden angegebenen Namen mit den Einträgen im Array *Person* vergleicht; wenn sie die Namen findet, werden in den beiden Records im Array *Bekannter* ein Pointer auf den jeweils anderen Record gesetzt, sonst werden vorher noch mit *registriere_neue_Person* neue Records angelegt). Wenn der Benutzer bei einem Namen keinen Buchstaben o.ä. eingibt, sondern nur die Eingabetaste drückt, wird damit das Einlesen der Namen der Partygäste beendet. Danach wird der Name der Person eingelesen, die von der Party hören soll (der „Robert") und für diese Person ebenfalls ein Record im Array *Person* angelegt und gleichzeitig noch der Pointer *Robert* auf diesen Record gesetzt (damit das Programm danach direkten Zugriff auf Roberts Daten hat); außerdem wird in diesem Record die Variable *von_Party_gehoert* auf 1 gesetzt, damit dort die Rekursion stoppt (sonst würde ja Robert selbst anderen von der Party erzählen). Danach werden in einer Endlosschleife die Bekanntschaften eingelesen und durch die Funktion *registriere_Bekanntschaft* in die Records eingetragen. Wie vorher wird aus der Endlosschleife ausgestiegen, wenn einer der beiden Namen leer ist (nur Eingabetaste).
Wenn alle Personen und Bekanntschaftsverhältnisse eingelesen sind, kann die Rekursion beginnen. Startpunkt dafür ist die erste Person im Array *Person*, da diese ein Partygast ist. Es werden ja zuerst alle Partygäste in das Array eingelesen, und wenn zumindest eine Person auf der Party war (was ja anzunehmen ist...), dann steht diese Person als erstes im Array. Mit Hilfe der Funktion weitersagen erzählt dieser Gast allen seinen Bekannten von der Party, d.h. bei allen seinen Bekannten wird die Variable *von_Party_gehoert* auf 1 gesetzt. Diese Funktion ruft sich selbst rekursiv mit allen Bekannten der als Parameter angegebenen Person auf. Damit die Rekursion nicht „im Kreis geht" d.h. Personen (evtl. über Umwege) sich selbst von der Party erzählen, wird den Leuten, die bereits von der Party gehört haben, es kein zweites mal erzählt. Konkret: bei Personen, bei denen bei Funktionsanfang die Variable *von_Party_gehoert* bereits auf 1 steht, stoppt die Rekursion und läßt die Bekannten dieser Person in Ruhe, da diese bereits informiert sein müssen. So würde die Rekursion den gesamten Bekanntenkreis der Partygäste informieren. Da aber das Stadium interessiert, in dem Robert von der Party erfährt, muss die Rekursion dort gestoppt werden, d.h.

Robert selbst darf niemandem von der Party erzählen, weil man sonst nicht mehr nachvollziehen kann, wer es nicht von Robert erfahren hat, sondern umgekehrt ihn informiert hat. Da ja die Rekursion bei einer Person gestoppt wird, die bereits von der Party erfahren hat, wird die Variable *von_Party_gehoert* bei Robert von Anfang an auf 1 gesetzt. Robert hat zwar in Wirklichkeit noch nichts von der Party gehört, aber es wird verhindert, dass Leute, die nur über Robert von der Party erfahren würden, von der Party wissen. Wenn die Rekursion fertig ist, wissen also alle Personen 'zwischen' dem ersten Partygast und Robert von der Party. Jetzt kann die Funktion *zaehle_Roberts_Bekannte* alle direkten Bekannten von Robert abfragen, ob sie von der Party gehört haben. Die positiven Antworten werden summiert − voilà, das ist die Lösung.

Halbformale Programmbeschreibung:

Lies die Namen der Partygäste ein und lege für jeden Namen einen Record an (im Array *Person*). Registriere dann für jeden Gast die Bekanntschaften zu allen anderen Gästen.

Lies den Namen der Person ein, die von der Party hören soll (="Robert") und setze *von_Party_gehoert* vom Record dieser Person auf 1 (damit später die Rekursion bei ihm stoppt).

Erfasse die Bekanntschaften: lies die Namen zweier Personen ein und registriere deren Bekanntschaft (Funktion *registriere_Bekanntschaft*).

Rekursion: Ermittle, beginnend beim ersten Partygast, mit Hilfe der Funktion *weitersagen*, wer alles von der Party erzählt bekommt.

Zähle nach, wieviele von Roberts direkten Bekannten von der Party wissen.

Programmende;

Funktion *weitersagen*(Parameter: Person, der von der Party erzählt wird):

Weiß die Person schon etwas von der Party, d.h. ist ihre Variable *von_Party_gehoert* = 1 ?
Wenn ja, dann braucht man ihr nicht mehr von der Party erzählen, sondern kann zur nächsten Person gehen.
Setze die Variable *von_Party_gehoert* auf 1 und rufe die Funktion rekursiv mit jedem ihrer Bekannten auf.

Funktionsende;

Funktion *registriere_Bekanntschaft*(1. Name, 2. Name):

Finde die Einträge zu diesen beiden Namen im Array *Person*;
Wenn bei einem oder beiden Namen noch kein Eintrag vorhanden ist, dann lege entsprechend neue Einträge an.
Richte in beiden Einträgen einen Pointer auf den jeweils anderen Eintrag ein.

Funktionsende;

Programmcode:

```
#include <stdio.h>                  // printf()
#include <string.h>                 // strcpy()
#include <conio.h>                  // getch()

#define MAX_PERSONEN          100    // maximal 100 Leute kennen sich
#define MAX_BEKANNTE          20     // eine Person kennt maximal 20 andere
                                     // Personen
#define MAX_NAME              15     // maximale Länge eines Namens + 1
                                     // in Buchstaben

struct TPerson
       {
        char Name[MAX_NAME];
        int von_Party_gehoert;
        int Anzahl_Bekanntschaften;
        TPerson *Bekannter[MAX_BEKANNTE];
       };

// Globale Variablen
TPerson Person[MAX_PERSONEN]; // die Gesamtheit aller beteiligten Personen
int Anzahl_Personen=0;              // Anzahl dieser Personen
TPerson *Robert=NULL;         // Pointer auf eine der Personen, die die
                      // Funktion des "Robert" hat
```

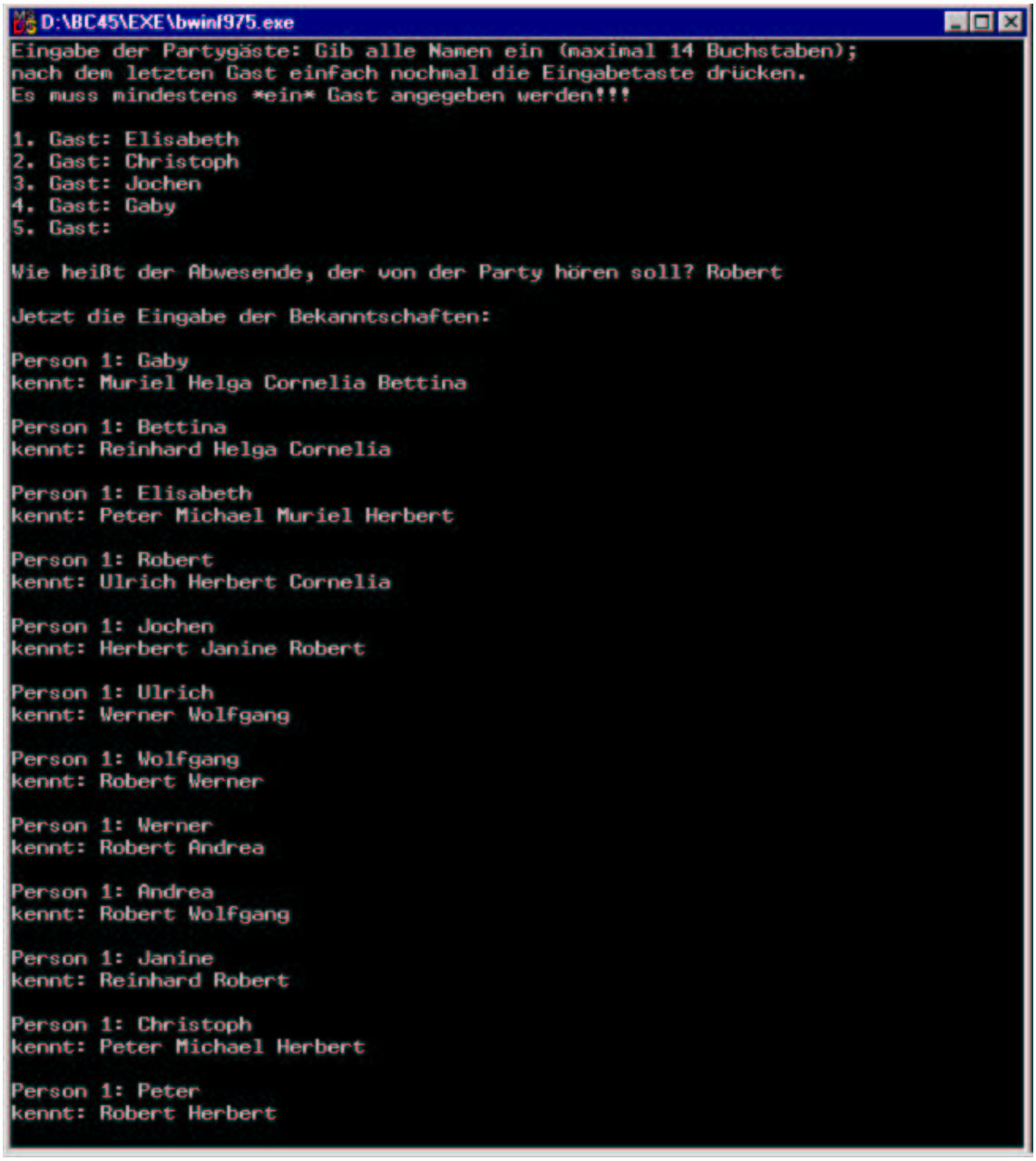
```
TPerson *registriere_neue_Person(const char *Name)
{
 Anzahl_Personen++;                  // eine Person mehr

 // in der Aufgabe ist ja ausdrücklich hingeschrieben, dass Katrin Käfer
 // niemandem etwas erzählt
 if(strcmp(Name,"Katrin")==0)
 {
  printf("Achtung: Falls Katrin Käfer gemeint ist: die darf nicht eingegeben"
        " werden,\nweil sie niemandem von der Party erzählt\n");
 }

 // hinter den letzten belegten Eintrag im Array 'Person' die Daten der neuen
 // Person eintragen
 strcpy(Person[Anzahl_Personen-1].Name,Name);
 Person[Anzahl_Personen-1].von_Party_gehoert=0;
 // neue Person - noch keine Bekanntschaften eingetragen
 Person[Anzahl_Personen-1].Anzahl_Bekanntschaften=0;

 // Pointer auf den neuen Record zurückgeben
 return &Person[Anzahl_Personen-1];
}

void registriere_Bekanntschaft(const char *Name1,const char *Name2)
{
 TPerson *Person_mit_Name1=NULL,*Person_mit_Name2=NULL;
 int i;// Schleifenzählvariable

 // durchsuchen, ob die beiden Namen schon bekannt sind
 for(i=0;i<Anzahl_Personen;i++)
 {
  if(strcmp(Name1,Person[i].Name)==0) Person_mit_Name1=&Person[i];
  if(strcmp(Name2,Person[i].Name)==0) Person_mit_Name2=&Person[i];
 }

 // falls Person mit dem eingegebenen Namen 'Name1' noch nicht registriert ist
 if(Person_mit_Name1==NULL) Person_mit_Name1=registriere_neue_Person(Name1);

 // falls Person mit dem eingegebenen Namen 'Name2' noch nicht registriert ist
 if(Person_mit_Name2==NULL) Person_mit_Name2=registriere_neue_Person(Name2);

 // Jetzt die Bekanntschaft registrieren (in beide Richtungen)
 Person_mit_Name1->Bekannter[Person_mit_Name1->Anzahl_Bekanntschaften++]=
       Person_mit_Name2;
 Person_mit_Name2->Bekannter[Person_mit_Name2->Anzahl_Bekanntschaften++]=
       Person_mit_Name1;
}

void weitersagen(TPerson *p)
{
 int i;                       // Schleifenzählvariable

 // wenn dieser Bekannte schon von der Party gehört hat, ist es unsinnig,
 // ihm davon zu erzählen; er weiß es ja schon und hat es schon allen seinen
 // Freunden gesagt
 if(p->von_Party_gehoert==1) return;

 // er wusste noch nichts von der Party, aber jetzt weiß er es!
 p->von_Party_gehoert=1;

 // jetzt sagt er es allen weiter (rekursiver Aufruf der Funktion)
 for(i=0;i<p->Anzahl_Bekanntschaften;i++)
 {
  weitersagen(p->Bekannter[i]);
 }
}

void gib_Ergebnis_aus()
{
 int Party_Meldungen=0;              // gibt an, wie oft Robert von der Party hört
 int i;               // Schleifenzählvariable

 printf("\n%s hört von...\n",Robert->Name);

 // bei allen seinen Bekannten wird die Variable 'von_Party_gehoert' abgefragt;
 // jedesmal, wenn sie auf 1 steht, hat dieser Bekannter von Katrins Party
 // gehört und wird es Robert erzählen, also wird 'Party_Meldungen' um 1 erhöht
 for(i=0;i<Robert->Anzahl_Bekanntschaften;i++)
 {
  if((Robert->Bekannter[i])->von_Party_gehoert==1)
  {
   printf("%s, ",Robert->Bekannter[i]->Name);
   Party_Meldungen++;
  }
 }

 printf("\nalso von %d Personen, wie toll Katrins Party war!\n",
       Party_Meldungen);
}

// zeige alle Bekanntschaftverhältnisse nochmal an, damit der arme Benutzer,
```

```
// der alle Namen einzeln eingeben musste, überprüfen kann, ob alles stimmt
void zeige_Freundeskreis()
{
 int i,j;
 char Klammer1,Klammer2;

 printf("\nZusammenfassung der Bekanntschaften\n(Klammern um einen Namen"
        " bedeuten, dass die entspr. Person nichts von der Party weiß):\n");

 for(i=0;i<Anzahl_Personen;i++)
 {
  printf("%s kennt ",Person[i].Name);

  for(j=0;j<Person[i].Anzahl_Bekanntschaften;j++)
  {

   if(Person[i].Bekannter[j]->von_Party_gehoert==1)
   {
    Klammer1=' ';
    Klammer2=' ';
   }
   else
   {
    Klammer1='(';
    Klammer2=')';
   }

   // auf einen ordentlichen Zeilenumbruch verzichte ich hier...
   printf("%c%s%c, ",Klammer1,Person[i].Bekannter[j]->Name,Klammer2);
  }

  printf("\n");
 }
}

// Hauptprogramm
int main()
{
 char Name1[MAX_NAME],Name2[MAX_NAME],Eingabe[MAX_BEKANNTE*MAX_NAME];
 char *String_Pointer;
 int i;               // Schleifenzählvariable

 printf("Eingabe der Partygäste: Gib alle Namen ein (maximal %d Buchstaben)"
       ";\nnach dem letzten Gast einfach nochmal die Eingabetaste drücken.\n"
       "Es muss mindestens *ein* Gast angegeben werden!!!\n\n",
       MAX_NAME-1);

 while(1)
 {

  printf("%d. Gast: ",Anzahl_Personen+1);
  gets(Name1);
  if(strcmp(Name1,"")==0) break;     // das war die letzte Eingabe

  registriere_neue_Person(Name1);

  // Bekanntschaft mit allen anderen Gästen registrieren
  for(i=0;i<Anzahl_Personen-1;i++)
  {
   registriere_Bekanntschaft(Name1,Person[i].Name);
  }

 }

 printf("\nWie heißt der Abwesende, der von der Party hören soll? ");
 gets(Name1);
 Robert=registriere_neue_Person(Name1);
 Robert->von_Party_gehoert=1; // damit bei ihm die Rekursion stoppt

 printf("\nJetzt die Eingabe der Bekanntschaften:\n");

 while(1)
 {

  printf("\nPerson 1: ");
  gets(Name1);
  // War das die letzte Eingabe ?
  if(strcmp(Name1,"")==0) break;

  printf("kennt: ");
  gets(Eingabe);
  // War das die letzte Eingabe ?
  if(strcmp(Eingabe,"")==0) break;

  // nach dem ersten Textstück suchen, das mit ein/mehreren Leerzeichen aufhört
  String_Pointer=strtok(Eingabe," ");

  // solange Textstücke gefunden werden
  while(String_Pointer!=NULL)
  {
   // ja - einen Namen gefunden => Gefundenes in die Variable Name2 kopieren
   strcpy(Name2,String_Pointer);
```

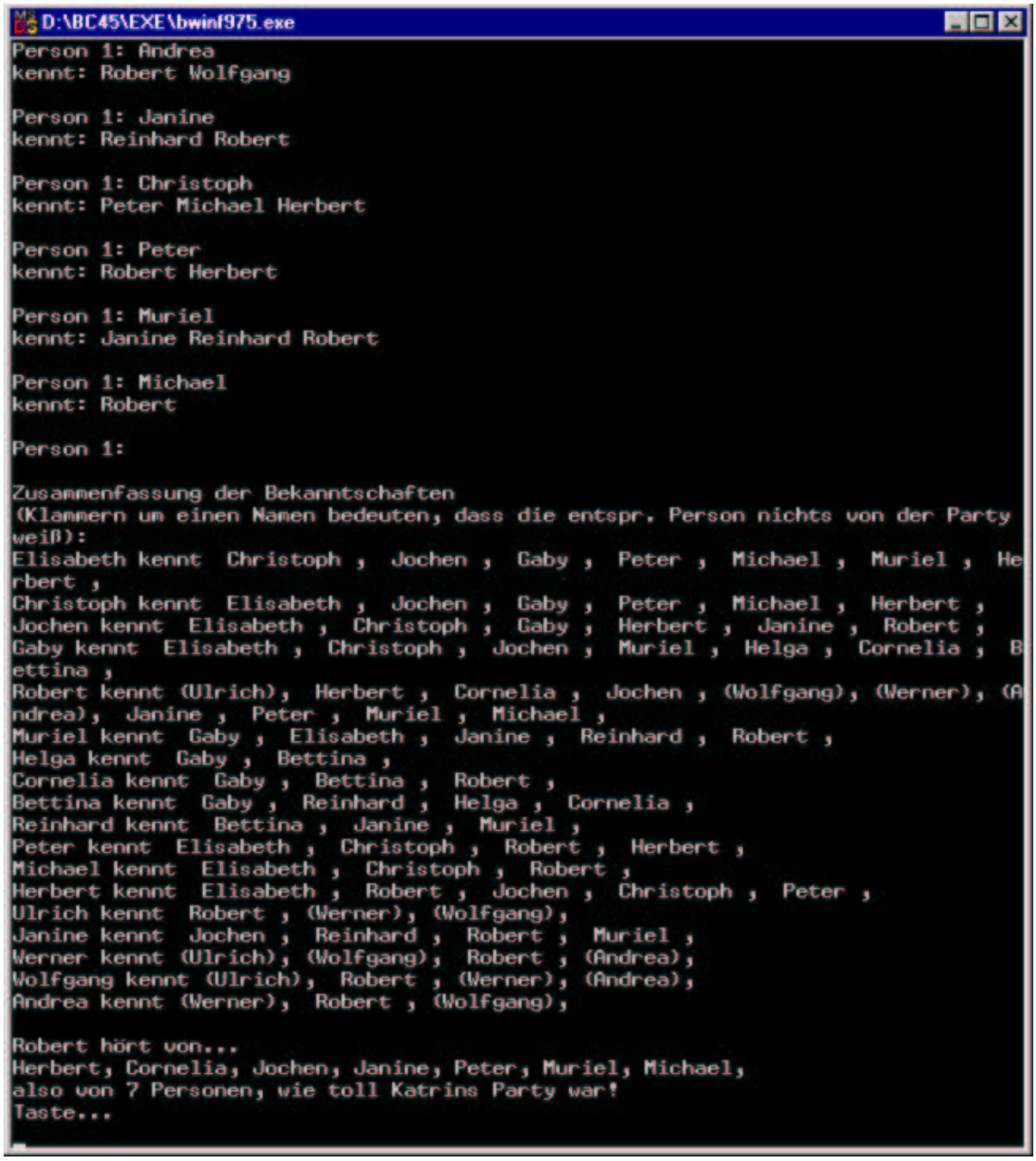
```
   // die Bekanntschaft sofort registrieren
   registriere_Bekanntschaft(Name1,Name2);

   // nach weiteren Namen suchen
   String_Pointer=strtok(NULL," ");  // sucht nach Leerzeichen
  }
 }

 // hier läuft die Rekursion ab, hier wird ermittelt, wer alles von der Party
 // erfährt (da ja mindestens ein Partygast vorhanden sein muss, damit es Sinn
 // macht, wird hier angenommen, dass dies so ist, also Person[0] der erste Gast
 // ist)
 for(i=0;i<Person[0].Anzahl_Bekanntschaften;i++) weitersagen(&Person[0]);

 // zeig mal kurz das Ergebnis der Rekursion...
 zeige_Freundeskreis();

 // ...und gib dann die Lösung aus
 gib_Ergebnis_aus();

 printf("Taste...\n");
 getch();

 return 0;
}
```

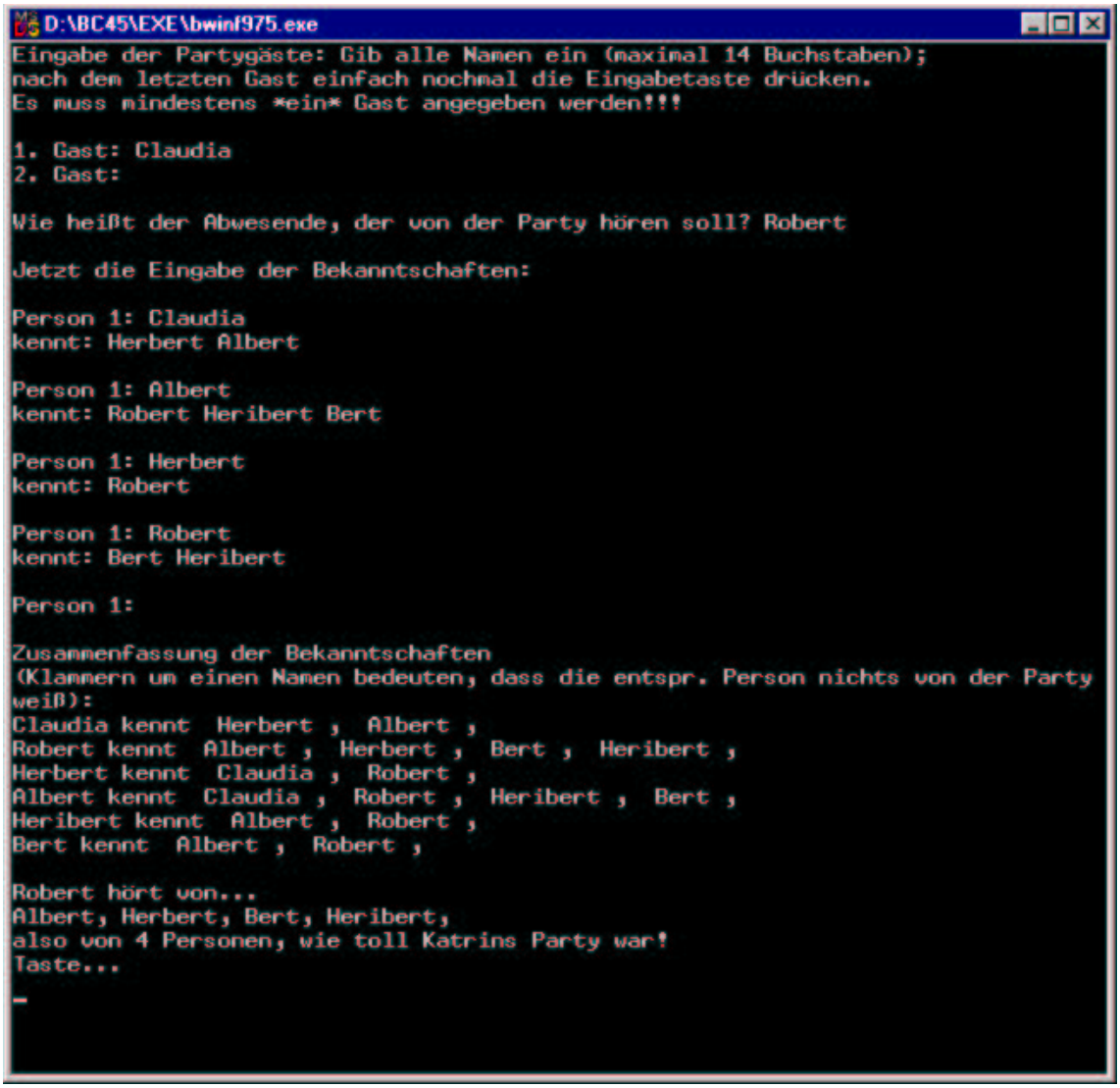
Programmablaufprotokoll (Screenshots):

Achtung, jetzt wird’s lang: die Lösung der Aufgabe

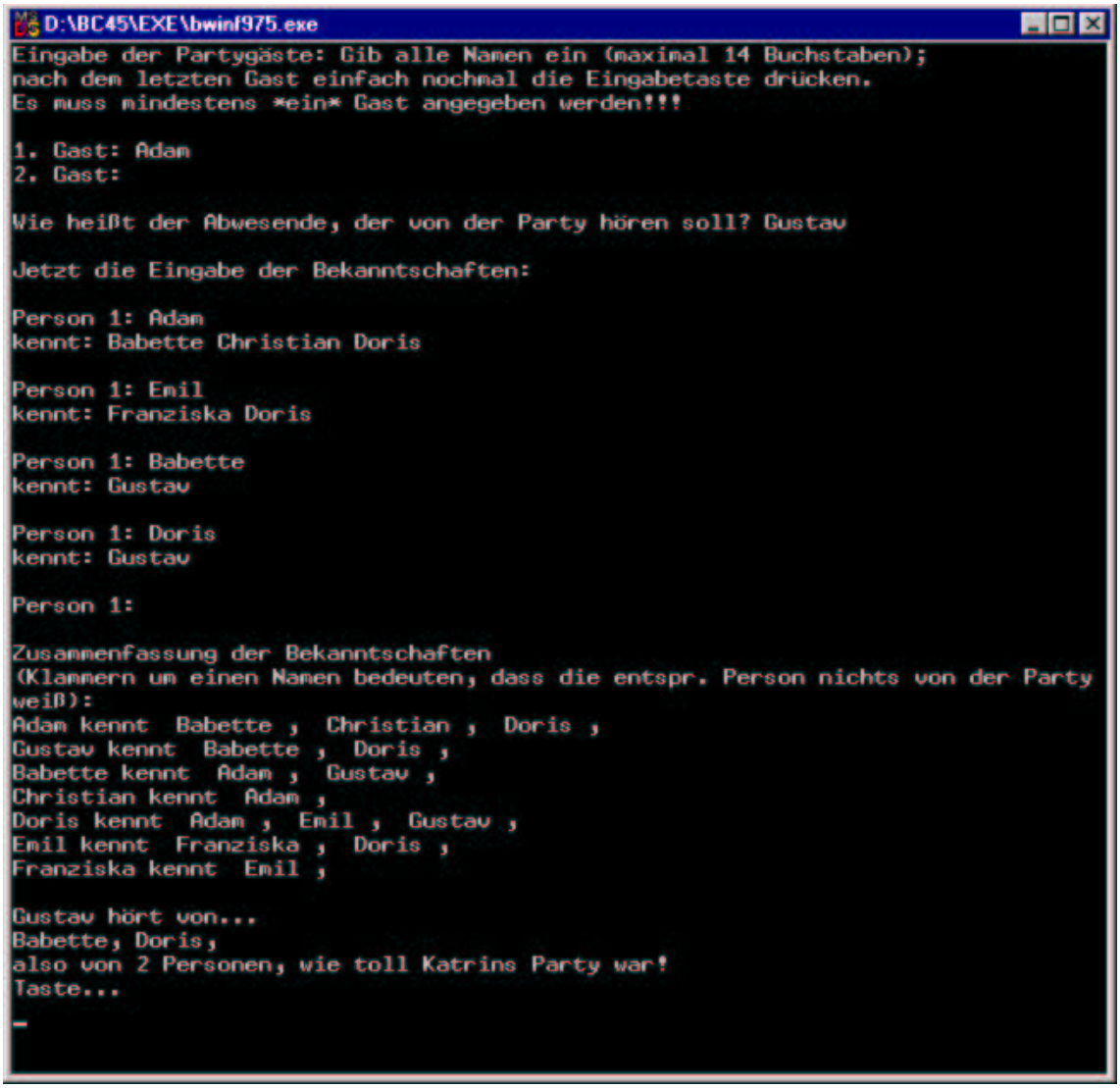


Fortsetzung nächste Seite


D:\BC45\EXE\bwinf975.exe
Person 1: Andrea
kennt: Robert Wolfgang
Person 1: Janine
kennt: Reinhard Robert
Person 1: Christoph
kennt: Peter Michael Herbert
Person 1: Peter
kennt: Robert Herbert
Person 1: Muriel
kennt: Janine Reinhard Robert
Person 1: Michael
kennt: Robert
Person 1:
Zusammenfassung der Bekanntschaften
(Klammern um einen Namen bedeuten, dass die entspr. Person nichts von der Party
weiß):
Elisabeth kennt Christoph , Jochen , Gaby , Peter , Michael , Muriel , He
rbert ,
Christoph kennt Elisabeth , Jochen , Gaby , Peter , Michael , Herbert ,
Jochen kennt Elisabeth , Christoph , Gaby , Herbert , Janine , Robert ,
Gaby kennt Elisabeth , Christoph , Jochen , Muriel , Helga , Cornelia , B
ettina ,
Robert kennt (Ulrich), Herbert , Cornelia , Jochen , (Wolfgang), (Werner), (A
ndrea), Janine , Peter , Muriel , Michael ,
Muriel kennt Gaby , Elisabeth , Janine , Reinhard , Robert ,
Helga kennt Gaby , Bettina ,
Cornelia kennt Gaby , Bettina , Robert ,
Bettina kennt Gaby , Reinhard , Helga , Cornelia ,
Reinhard kennt Bettina , Janine , Muriel ,
Peter kennt Elisabeth , Christoph , Robert , Herbert ,
Michael kennt Elisabeth , Christoph , Robert ,
Herbert kennt Elisabeth , Robert , Jochen , Christoph , Peter ,
Ulrich kennt Robert , (Werner), (Wolfgang),
Janine kennt Jochen , Reinhard , Robert , Muriel ,
Werner kennt (Ulrich), (Wolfgang), Robert , (Andrea),
Wolfgang kennt (Ulrich), Robert , (Werner), (Andrea),
Andrea kennt (Werner), Robert , (Wolfgang),
Robert hört von...
Herbert, Cornelia, Jochen, Janine, Peter, Muriel, Michael,
also von 7 Personen, wie toll Katrins Party war!
Taste...


D:\BC45\EXE\bwinf975.exe
Eingabe der Partygäste: Gib alle Namen ein (maximal 14 Buchstaben);
nach dem letzten Gast einfach nochmal die Eingabetaste drücken.
Es muss mindestens *ein* Gast angegeben werden!!!
1. Gast: Claudia
2. Gast:
Wie heißt der Abwesende, der von der Party hören soll? Robert
Jetzt die Eingabe der Bekanntschaften:
Person 1: Claudia
kennt: Herbert Albert
Person 1: Albert
kennt: Robert Heribert Bert
Person 1: Herbert
kennt: Robert
Person 1: Robert
kennt: Bert Heribert
Person 1:
Zusammenfassung der Bekanntschaften
(Klammern um einen Namen bedeuten, dass die entspr. Person nichts von der Party
weiß):
Claudia kennt Herbert , Albert ,
Robert kennt Albert , Herbert , Bert , Heribert ,
Herbert kennt Claudia , Robert ,
Albert kennt Claudia , Robert , Heribert , Bert ,
Heribert kennt Albert , Robert ,
Bert kennt Albert , Robert ,
Robert hört von...
Albert, Herbert, Bert, Heribert,
also von 4 Personen, wie toll Katrins Party war!
Taste...

1. Beispiel


D:\BC45\EXE\bwinf975.exe
Eingabe der Partygäste: Gib alle Namen ein (maximal 14 Buchstaben);
nach dem letzten Gast einfach nochmal die Eingabetaste drücken.
Es muss mindestens *ein* Gast angegeben werden!!!
1. Gast: Adam
2. Gast:
Wie heißt der Abwesende, der von der Party hören soll? Gustav
Jetzt die Eingabe der Bekanntschaften:
Person 1: Adam
kennt: Babette Christian Doris
Person 1: Emil
kennt: Franziska Doris
Person 1: Babette
kennt: Gustav
Person 1: Doris
kennt: Gustav
Person 1:
Zusammenfassung der Bekanntschaften
(Klammern um einen Namen bedeuten, dass die entspr. Person nichts von der Party
weiß):
Adam kennt Babette , Christian , Doris ,
Gustav kennt Babette , Doris ,
Babette kennt Adam , Gustav ,
Christian kennt Adam ,
Doris kennt Adam , Emil , Gustav ,
Emil kennt Franziska , Doris ,
Franziska kennt Emil ,
Gustav hört von...
Babette, Doris,
also von 2 Personen, wie toll Katrins Party war!
Taste...

2. Beispiel

3. Beispiel

Anmerkungen:

- Das Programm unternimmt praktisch keine Überprüfung der Eingaben; der Benutzer muss also selbst darauf achten, dass er gültige Werte eingibt und auch z.B. keine Namen oder Bekanntschaften mehrfach eingibt.

- Die maximale Anzahl an Personen ist durch die Konstante *MAX_PERSONEN* auf 100 festgelegt; die maximale Zahl von Bekannten pro Person ist durch *MAX_BEKANNTE* auf 20 festgelegt; die maximale Länge des Namens der Personen ist durch *MAX_NAME* auf 14 Buchstaben festgelegt; werden bei der Dateneingabe diese Grenzen überschritten, tritt keine Fehlermeldung auf, aber das Verhalten des Programms ist undefiniert! Allerdings können diese Konstanten je nach Bedarf angepasst werden.
- Die Personen werden über die Namen identifiziert. Also muss der Benutzer für die gleiche Person den Namen immer 100%ig gleich schreiben (incl. Groß−/Kleinschreibung und Leerzeichen o.ä.); also ist ’Robert’ eine andere Person als ’ROBERT’; Nachnamen sind auch nicht erlaubt, weil sie als 2 Namen verstanden werden (wg. Leerzeichen)

Teilnahmebescheinigung

Fehlerbericht

Urkunde

Literaturverzeichnis

- Heyderhoff, P. (Hrsg.), Bundeswettbewerb Informatik, Aufgaben und Lösungen, Band 1, Stuttgart, Ernst−Klett−Verlag, 1989
- Internet: http://www.bwinf.de, 1997−1998
- Sedgewick, R., Algorithmen in C++, Bonn, Addison−Wesley−Verlag, 1992, S. 125 f

Bestätigung der selbständigen Anfertigung

Ich erkläre hiermit, dass ich die Facharbeit ohne fremde Hilfe angefertigt habe und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt habe.

....., den ....
Ort Datum Unterschrift des Schülers