GYMNASIUM GRAFING

KOLLEGSTUFE 1996/1998

Leistungskurs Mathematik

Facharbeit

Thema: Bundeswettbewerb Informatik 97/98, 1.Runde

Verfasser: Michael Wack

Kursleiter: StD Christine Vorbach

Abgabetermin: 2.2.98

Bewertung: Note: _____ Punkte: _____

Unterschrift des Kursleiters: _____

Inhaltsverzeichnis

Der Bundeswettbewerb Informatik

Der Bundeswettbewerb Informatik findet jährlich statt. In diesem Jahr zum 16. Mal. Der Wettbewerb beginnt und endet im September, dauert etwa ein Jahr und besteht aus drei Runden. In der ersten und zweiten Runde sind fünf bzw. drei Aufgaben zu Hause selbständig zu bearbeiten. Die Bearbeitungszeit beträgt circa zwei Monate. An der zweiten Runde dürfen jene, die wenigstens drei Aufgaben weitgehend richtig gelöst haben, teilnehmen. Die Bewertung erfolgt durch eine relative Plazierung der Arbeiten. Die ca. dreißig bundesweit Besten werden zur dritten Runde, einem Kolloquium, eingeladen. Darin führt jeder ein Gespräch mit je einem Informatiker aus Schule und Hochschule und analysiert bzw. bearbeitet im Team zwei Informatik-Probleme.

Die Teilnehmerzahlen haben sich seit vergangenem Jahr sehr erfreulich entwickelt. Die Beteiligung erhöhte sich von 960 auf 1255 Schülerinnen und Schüler. Auch der Anteil an Mädchen stieg von 2.9% auf 3,2%.

Nähere Informationen zum Bundeswettbewerb sowie Teilnehmerstatistiken lassen sich unter der Internetadresse http://www.bwinf.de abfragen.

Allgemeine Hinweise

Alle Programme, die zur Lösung der gestellten Aufgaben nötig waren, wurden mit Borland C++ 4.5 geschrieben und getestet. Um den Quelltext möglichst einfach zu gestalten, wurden sie als EasyWin-Applikationen kompiliert. Nur das Programm zum Zeichnen der „Rekursiven Besen“ aus Aufgabe 3 wurde als reines DOS-Programm kompiliert, da bei EasyWin-Applikationen keine Grafikausgabe möglich ist.
Im Quell-Text verwendete Kommentare beziehen sich immer auf die direkt links daneben oder genau darunter befindlichen Anweisungen. In der Programm-Dokumentation sind alle Ausdrücke, die exakt aus dem Programm-Text übernommen wurden, *kursiv* dargestellt. Die einzelnen Seiten sind für alle Aufgaben durchlaufend numeriert.
Bei der Eingabe von Zahlen ist bei allen Programmen grundsätzlich zu beachten, daß diese intern als Integer-Typen gehandhabt werden. Deshalb sollten keine Zahlen außerhalb des zulässigen Bereichs von -32768 bis 32767 eingegeben werden.

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:

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

Lösung von Aufgabe 1: Belagerung um Farnsworth Castle

1. Welchen Anforderungen müssen Anzahl und Anordnung der Löcher genügen, damit die Schablone zur Verschlüsselung geeignet ist?

Damit eine bestimmte Schablone zur Verschlüsselung geeignet ist, darf der zu verschlüsselnde Text maximal vier mal so viele Buchstaben, wie die Schablone Löcher, haben. Sonst wäre der Text noch nicht komplett bearbeitet, wenn die Schablone zum vierten Mal um 90° gedreht werden müßte. Eine weitere Drehung würde zum Überschreiben der als erstes verschlüsselten Buchstaben führen.
Außerdem sollten sich die Positionen der Löcher nicht überschneiden, wenn die Schablone gedreht wird. D.h. bei jeder Drehung um 90° dürfen die Löcher der Schablone nicht wieder auf Positionen zu liegen kommen, auf denen sich bereits bei einer der vorhergehenden Drehungen ein Loch befand.

2. Gib ein einfaches Verfahren für den Entwurf von geeigneten Schablonen an.

Um möglichst einfach und sicher eine zum Verschlüsseln von Texten geeignete Schablone zu erhalten, geht man wie folgt vor:
Als Erstes entscheidet man sich auf Grund des zu verschlüsselnden Textes für eine geeignete Schablonengröße. Die Schablone muß insgesamt soviele verfügbare Felder wie der Text Buchstaben haben. Bei einer ungeraden Seitenlänge der Schablone muß es ein Feld mehr wie zu verschlüsselnde Buchstaben geben, da das Feld in der Mitte nicht als Loch benutzt werden kann, weil es seine Position bei einer Drehung nicht verändert. Will man beispielsweise einen Text mit 16 Buchstaben verschlüsseln, braucht man eine Schablone mit einer Seitenlänge von vier Feldern, da $4^2 = 16$. Für einen Text mit 25 Buchstaben benötigt man allerdings bereits eine Seitenlänge von 6 Feldern, da bei einer Größe von 5 * 5 Feldern nur 24 Buchstaben verschlüsselt werden könnten, da das mittlere Feld ja nicht benutzt werden kann !
Zur eigentlichen Erstellung der Schablone sucht man sich ein beliebiges, noch nicht belegtes Feld aus und kennzeichnet es z.B. mit einem L (=Loch). Anschließend markiert man alle Felder, auf die sich das soeben gekennzeichnete Loch durch Drehung verschieben läßt, mit einem anderen Symbol (im Beispiel B). Dieses Vorgehen wiederholt man nun bis man die benötigte Anzahl von Löchern hat oder alle Felder verbraucht sind. Bei Schablonen mit ungeraden Seitenlängen muß man allerdings darauf achten, das mittlere Feld nicht zu verwenden. Wenn man nun die mit L gekennzeichneten Felder ausschneidet, erhält man die endgültige Schablone.

Beispiel:

1. Schritt

L				B
B				B

⇒

2. Schritt

L				B
		L		
	B		B	
		B		
B				B

⇒

3. Schritt

L			L	B
B		L		
	B		B	
		B		B
B	B			B

4. Schritt

L		B	L	B
B		L		
B	B		B	B
		B		B
B	B	L		B

⇒

5. Schritt

L		B	L	B
B	B	L	B	
B	B		B	B
	B	B	L	B
B	B	L		B

⇒

6. Schritt

L	B	B	L	B
B	B	L	B	L
B	B		B	B
B	B	B	L	B
B	B	L	B	B

3. Schreibe ein Programm, das anhand einer gegebenen Schablone eine Nachricht ver- und entschlüsseln kann.

3.1 Lösungsidee

Um eine Nachricht ver- oder entschlüsseln zu können muß man zunächst eine geeignete Möglichkeit finden, das Layout, d.h. die Positionen der Löcher einer Schablone zu speichern. Dies geschieht in meinem Programm in einem zweidimensionalen Array von Zeichen *[Im Programm: Schablonen-Layout]*. Dies vereinfacht das Einlesen der Schablone, aber auch das "Drehen" der Schablone läßt sich auf diese Weise relativ simpel realisieren. Es wird nämlich nicht die Schablone an sich gedreht, sondern die einzelnen Felder werden, je nach Drehrichtung, in verschiedenen Reihenfolgen durchlaufen. Der genaue Ablauf ist in 3.2 näher erläutert. Soll nun eine Nachricht verschlüsselt werden, so werden die einzelnen Felder der Schablone nacheinander überprüft, ob sie ausgeschnitten sind. In diesem Fall wird immer genau ein Buchstabe der zu verschlüsselnden Nachricht *[Im Programm: Nachricht]* an die entsprechende Stelle in einem weiteren, ebenfalls zweidimensionalen Array der selben Dimension wie die Schablone *[Im Programm: VerschlNachricht]* kopiert. Dieser Vorgang wird so lange wiederholt, bis die Schablone in allen vier möglichen Richtungen durchlaufen wurde. Danach befindet sich die verschlüsselte Nachricht bereits in ihrer endgültigen Fassung in *VerschlNachricht*. Das Entschlüsseln einer Nachricht läuft genauso ab, nur daß bei jedem ausgeschnittenen Feld der Buchstabe, der sich an der entsprechenden Position in *VerschlNachricht* befindet, an *Nachricht* angehängt wird und sich der entschlüsselte Text deshalb am Ende in *Nachricht* befindet.

3.2 Programm-Dokumentation

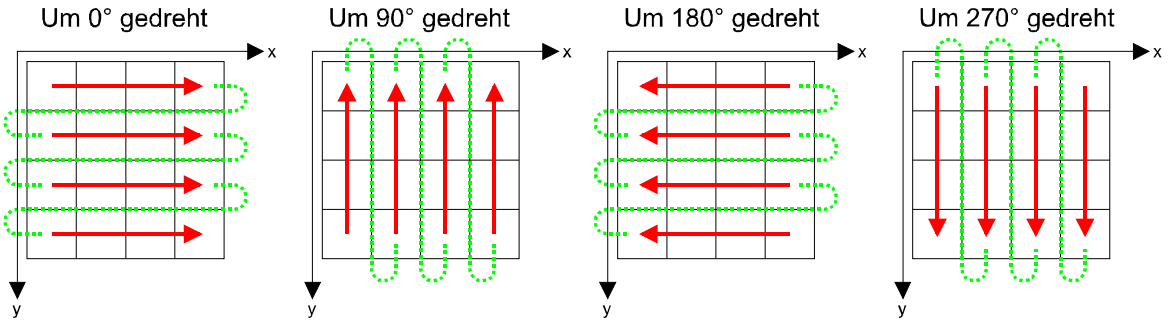
Am Anfang des Programms wird zunächst der Datentyp bool sowie seine beiden Werte true und false definiert, um das Programm übersichtlicher zu gestalten. Anschließend werden einige globale Variablen deklariert, die hier einzeln erläutert sind:

SchablonenSize	beinhaltet die Seitenlänge der Schablone (wird vom Benutzer eingegeben)
SchablonenLayout	beinhaltet das Layout, d.h. die Positionen von ausgeschnittenen Feldern einer Schablone. *SchablonenLayout* zeigt auf ein Array von Strings. Diese Strings beinhalten die einzelnen Zeilen der Schablone, wie sie vom Benutzer eingegeben werden. Für ausgeschnittene Felder ein O, sonst ein X.
VerschlNachricht	nimmt die verschlüsselte Nachricht auf. Entweder als Ergebnis einer Verschlüsslung oder als Eingabe des Benutzers. Sie ist im selben Format wie *SchablonenLayout* gespeichert, nur daß sie natürlich nicht X und O beinhaltet, sondern die Buchstaben des verschlüsselten Textes.
Nachricht	beinhaltet die unverschlüsselte Nachricht

Nun folgen die Funktionsköpfe der Funktionen, die dann weiter unten implementiert sind. Das Hauptprogramm gestaltet sich denkbar einfach. Nachdem durch *Init()* die globalen Variablen mit 0 initialisiert wurden, wartet *Menu()* auf die Eingabe eines 'V' oder eines 'E' je nachdem ob ver- oder entschlüsselt werden soll und gibt dementsprechend 1,2 oder bei einer ungültigen Eingabe 0 zurück. Mit Hilfe einer *switch()*-Anweisung wird die Funktion *Verschlüsseln(bool bVerschl)* mit *bVerschl = true* aufgerufen, wenn ein Text verschlüsselt oder mit *bVerschl = false*, wenn ein Text entschlüsselt werden soll. Am Ende des Hauptprogramms wird der belegte Speicher durch die Funktion *Loesche-Schablone()* wieder freigegeben.

Die Funktion *LeseVerschlNachricht()* nimmt eine verschlüsselte Nachricht vom Benutzer entgegen, wobei der Speicher für *VerschlNachricht* bereits reserviert sein muß, sonst gibt die Funktion den Fehlercode 1 zurück. Das Einlesen geschieht in einer *for*-Schleife, die am Anfang jeder Zeile die Zeilennummer ausgibt und anschließend mit *gets()* eine Zeile einliest. Hier sollte bei der Eingabe

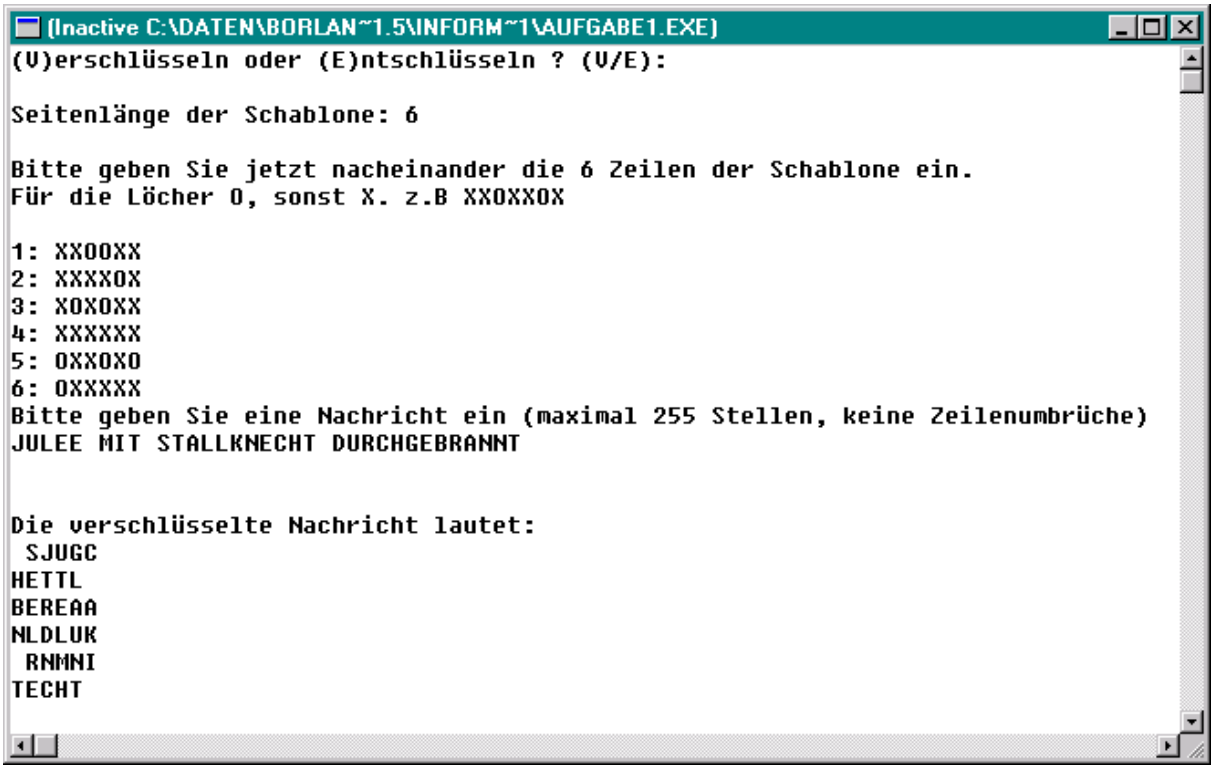
nach Möglichkeit keine Zeilenlänge die Seitenlänge der Schablone übersteigen, da ansonsten der String *VerschlNachricht[Zeile]* zu klein ist und es zum Absturz des Programms kommen kann.
Die Funktion *LeseSchablone()* arbeitet nach demselben Prinzip, nur daß hier noch der Speicher für *SchablonenLayout* und *VerschlNachricht* mit *new* reserviert und initialisiert wird. Außerdem liest das Programm am Anfang der Funktion noch die Seitenlänge der Schablone ein. Auch hier sollte man bei der Eingabe der Schablone die maximale Länge nicht überschreiten. Außerdem ist zu beachten, daß die Schablone, um das Programm übersichtlich zu halten, nicht auf ihre Richtigkeit hin überprüft wird. Es muß daher eine gültige, am besten nach dem Verfahren, das in Punkt 2 dieses Dokuments vorgestellt wurde, erstellte Schablone eingegeben werden. Andernfalls kommt es zwar zu keinem Absturz, aber das Ergebnis ist dann natürlich nicht sinnvoll. Ebenfalls nicht abgefangen wird der Fall, daß etwas anderes als 'X' oder 'O' eingegeben wird. Da das Programm lediglich nach 'O' sucht, wirkt sich eine Eingabe ohne Felder mit 'O' wie eine Schablone ohne Löcher hat aus. Es wird also nichts ver- oder entschlüsselt.
In der Funktion *LoescheSchablone()* wird der reservierte Speicher mit Hilfe von *delete* wieder freigegeben.
Zur Funktion *Menu()* läßt sich nicht viel sagen. Sie gibt lediglich bei der Eingabe von 'V' oder 'v' 1, bei 'E' oder 'e' 2 und sonst 0 zurück.
Das Hauptaugenmerk sollte sich auf die Funktion *Verschlüsseln(bool bVerschl)* richten, da hier das eigentliche Ver- bzw. Entschlüsseln stattfindet. Zu Beginn dieser Funktion wird je nach gewünschtem Vorgang eine verschlüsselte oder eine unverschlüsselte Nachricht sowie das Layout der Schablone eingelesen. Soll ein Text verschlüsselt werden, wird zusätzlich überprüft, ob die Anzahl der Löcher der Schablone ausreichen, um den kompletten Text zu verschlüsseln. Hierzu wird die Anzahl der Löcher zunächst gezählt und anschließend überprüft, ob die Nachricht mehr als vier mal so viele Buchstaben wie die Schablone Löcher enthält. Ist dies der Fall so wird eine Fehlermeldung ausgegeben und das Programm abgebrochen. Nun folgen vier Blöcke von jeweils zwei ineinander liegenden *for*-Schleifen, die das Durchlaufen der Schablone, entsprechend ihrer momentanen Ausrichtung, in der richtigen Reihenfolge übernehmen. Im einzelnen ergeben sich folgende Abläufe (am Beispiel einer Schablone der Seitenlänge vier):



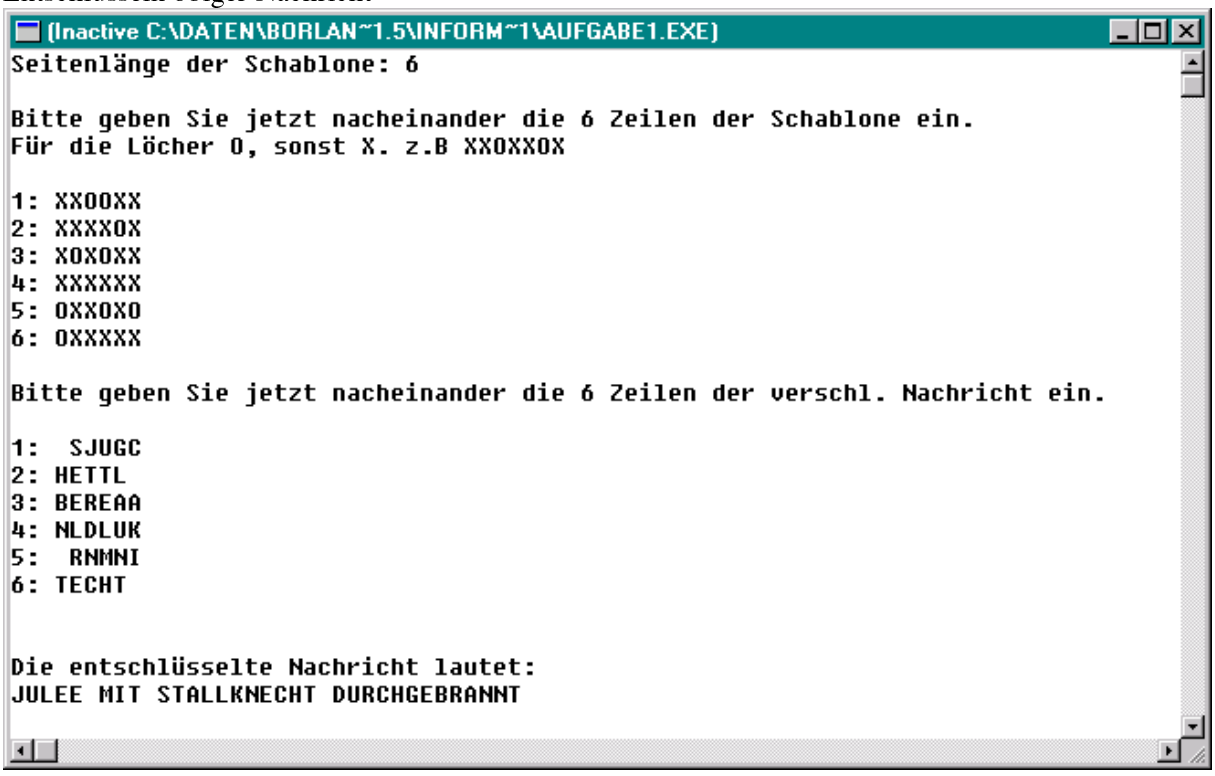
In der jeweils inneren *for*-Schleife, die alle Felder der Schablone durchläuft, wird überprüft, ob das entsprechende Feld ein 'O' enthält, also ausgeschnitten ist. In diesem Fall wird festgestellt, ob ver- oder entschlüsselt werden soll. Im ersten Fall wird der nächste, noch nicht bearbeitete Buchstabe aus *Nachricht* an die der Ausrichtung der Schablone entsprechende Stelle in *VerschlNachricht* kopiert. Im zweiten Fall wird der Buchstabe, der sich an der entsprechenden Stelle in *VerschlNachricht* befindet, an *Nachricht* angehängt. Dabei wird beim Verschlüsseln durch die Variable *c*, die stets die Anzahl der bereits bearbeiteten Zeichen beinhaltet, sichergestellt, daß die Schleife verlassen wird, sobald *c* den Wert der Variablen *len*, welche die Länge der nicht verschlüsselten Nachricht enthält, erreicht. Am Ende der Funktion wird dann noch das Ergebnis, d.h die verschlüsselte bzw. entschlüsselte Nachricht ausgegeben.

3.3 Programmablauf-Protokoll

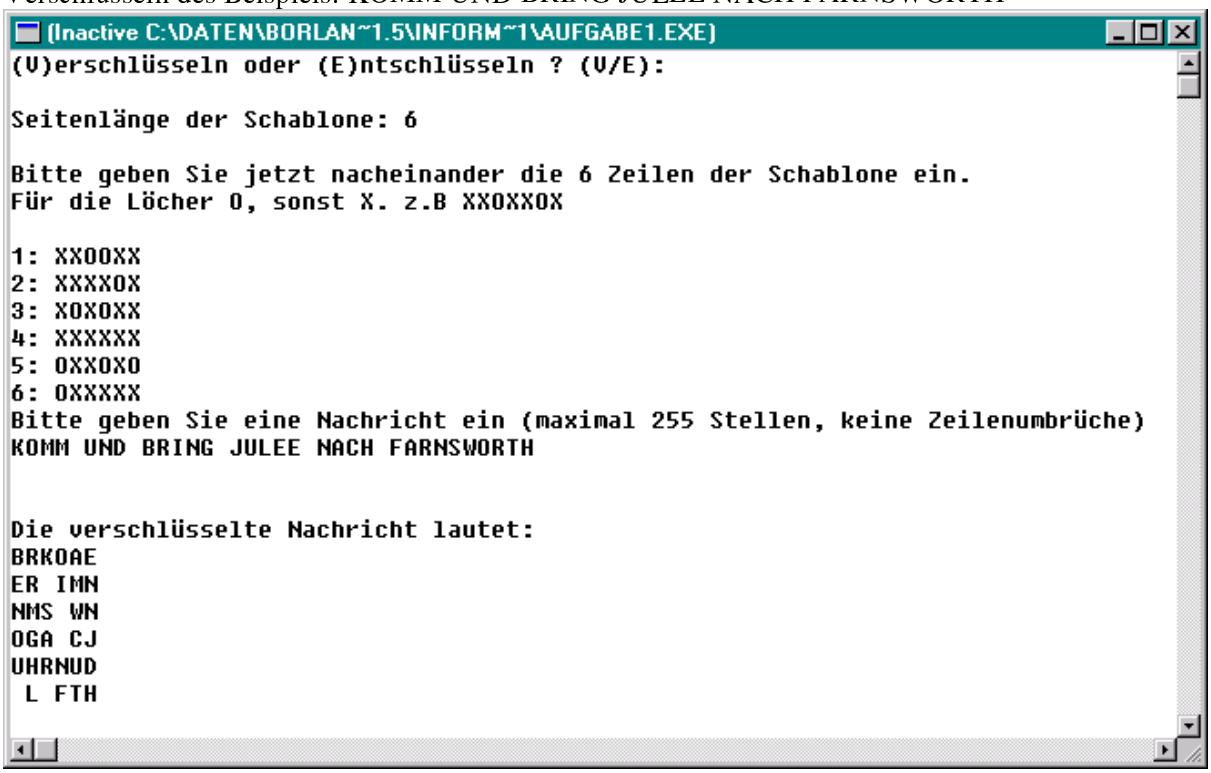
Verschlüsseln der Nachricht: JULEE MIT STALLKNECHT DURCHGEBRANNT



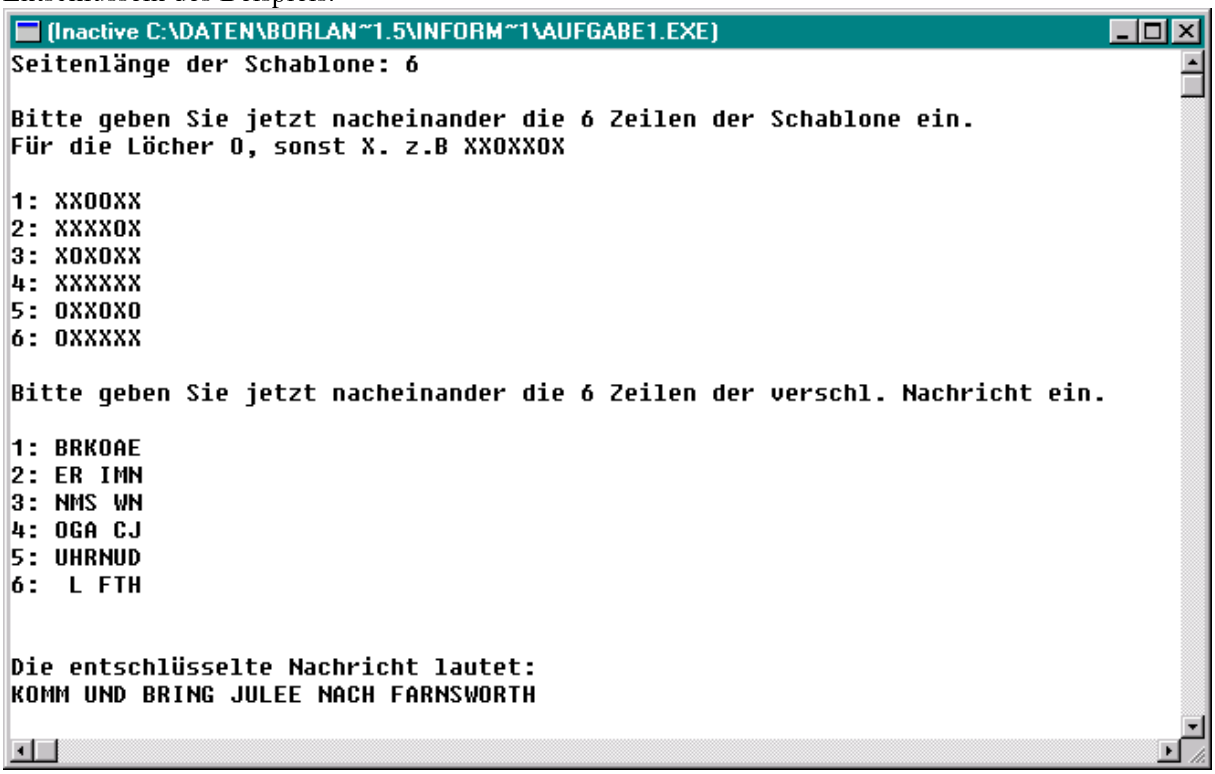
Entschlüsseln obiger Nachricht



Verschlüsseln des Beispiels: KOMM UND BRING JULEE NACH FARNSWORTH



Entschlüsseln des Beispiels:



3.4 Programm-Text

```
// Informatik-Wettbewerb
// Aufgabe 1
// Belagerung um Farnworth Castle
// (c) Michael Wack 1997
// 254 lines

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>

#define    true        1    // Definition des Datentyps bool
#define    false       0    // einschließlich seiner beiden
#define    bool        int  // möglichen Werte true, false.

// Globale Variablen
// SchablonenSize enthält die Seitenlänge der Schablone
int        SchablonenSize;

// Zeiger auf ein Array von Strings der Dimension SchablonenSize * SchablonenSize
// In jedem String ist eine Zeile der Schablone im Format O:Loch, X:kein Loch gespeichert
char       **SchablonenLayout;

// Zeiger auf ein Array von Strings der Dimension SchablonenSize * SchablonenSize
// Die Strings enthalten die einzelnen Zeilen der verschlüsselte Nachricht
char       **VerschlNachricht;

// Nachricht ist ein normaler String der die nicht verschl. Nachricht aufnimmt
char       Nachricht[ 255];


// Funktionen
void       Init();                      // Setzt einige Standardwerte
int        LeseVerschlNachricht();      // Liest eine verschl. Nachricht ein
int        LeseSchablone();             // Liest eine Schablone, sowie deren Größe ein.
void       LoescheSchablone();          // Gibt den von SchablonenLayout und VerschlNachricht
                                        // belgten Speicher frei
int        Menu();                      // Menu zur Auswahl ob ver- oder entschlüsselt werden soll

// Verschlüsselt oder Entschlüssel eine Nachricht in Abhängigkeit von bVerschl
void       Verschluesseln( bool bVerschl);

void Init()
{
           memset( Nachricht, 0, 255);
           SchablonenSize = 0;
           SchablonenLayout = NULL;
           VerschlNachricht = NULL;
}

int LeseVerschlNachricht()
{
           int Zeile;
           if ( !VerschlNachricht) return 1;
           printf( "\n\rBitte geben Sie jetzt nacheinander die %d Zeilen der verschl. Nachricht ein.\n\n", SchablonenSize);

           for (Zeile = 0; Zeile < SchablonenSize; Zeile++) // Zeilenweise die verschl. Nachricht lesen
           {
                      printf( "%d: ", Zeile+1); gets( VerschlNachricht[ Zeile]);
           }
           return 0;
}
```

```
int LeseSchablone()
{
	int res, c;

	LoescheSchablone();  // Falls bereits Speicher für eine Schablone oder verschl. Nachricht reserviert wurde
			     // wird dieser hier wieder frei gegeben
	printf( "Seitenlänge der Schablone: ");
	res = scanf( "%d", &SchablonenSize);

	if( res == 0 || res == EOF || SchablonenSize <= 0)
	{
		printf( "\r\nFehler bei der Eingabe !");
		return 1;
	}
	// Speicher reservieren
	SchablonenLayout = new char*[ SchablonenSize];
	VerschlNachricht = new char*[ SchablonenSize];

	printf( "\n\rBitte geben Sie jetzt nacheinander die %d Zeilen der Schablone ein.", SchablonenSize);
	printf( "\n\rFür die Löcher O, sonst X. z.B XXOXXOX\n\n");

	for (c = 0; c < SchablonenSize; c++) // Jede Zeile der Schablone einzeln einlesen
	{
		VerschlNachricht[ c] = new char[ SchablonenSize + 1]; // Speicher für diese Zeile anfordern
		memset( VerschlNachricht[ c], ' ', SchablonenSize);     // mit Leerzeichen initialisieren
		VerschlNachricht[c][SchablonenSize] = 0;                // mit 0 terminieren
		SchablonenLayout[ c] = new char[ SchablonenSize + 1]; // Speicher für diese Zeile anfordern
		printf( "%d: ", c+1); scanf("%s", SchablonenLayout[ c]); // SchablonenLayout-Zeile einlesen
	}
	return 0;
}

void LoescheSchablone()
{
	int c;
	if ( SchablonenSize != 0 && SchablonenLayout != NULL)
	{  // Schablonenlayout löschen
		for (c = 0; c < SchablonenSize; c++)
		{
			delete [] SchablonenLayout[ c];
		}
		delete [] SchablonenLayout;

		SchablonenSize = 0;
		SchablonenLayout = 0;
	}

	if ( SchablonenSize != 0 && VerschlNachricht != NULL)
	{  // VerschlNachricht löschen
		for ( c = 0; c < SchablonenSize; c++)
		{
			delete [] VerschlNachricht[ c];
		}
		delete [] VerschlNachricht;

		VerschlNachricht = NULL;
	}
}

int Menu()
{
	char ch;
	printf( "\r\n(V)erschlüsseln oder (E)ntschlüsseln ? (V/E): ");
	ch = getch();
	printf( "\n\n\r");
	if( ch == 'v' || ch == 'V') return 1;		// Verschlüsseln
	else if( ch == 'e' || ch == 'E') return 2;	// Entschlüsseln
	else return 0;
}
```

```
void Verschluesseln( bool bVerschl) // bVerschl = true  -> Verschlüsseln
{                                   // bVerschl = false -> Entschlüsseln
        int x, y, c, len;

        if ( LeseSchablone()) return;   // SchablonenLayout einlesen
        if( bVerschl)                   // falls Verschlüsselt werden soll, unverschl. Nachricht einlesen
        {
                printf( "Bitte geben Sie eine Nachricht ein (maximal 255 Stellen, keine Zeilenumbrüche)\n\r");
                gets( Nachricht);

                len = strlen( Nachricht);
                if ( len <= 0)
                {
                        printf( "Fehler bei der Eingabe !");
                        return;
                }

                // In den beiden folgenden for-Schleifen werden die Löcher in der Schablone gezählt.
                c = 0;
                for( y = 0; y < SchablonenSize; y++)
                {
                        for( x = 0; x < SchablonenSize; x++)
                        {
                                if( SchablonenLayout[y][x] == 'O') c++;
                        }
                }

                // Hat der Text mehr als 4 mal soviele Stellen wie die Schablone Löcher hat,
                // kann er nicht vollständig verschlüsselt werden und es erfolgt die Ausgabe
                // einer Fehlermeldung
                if( c*4 < len)
                {
                        printf( "Der Text ist zu lang für diese Schablone.\n");
                        return;
                }
        }
        else if (LeseVerschlNachricht()) return; // sonst verschl. Nachricht einlesen

        c = 0;

        /* 1. Richtung: Die Schablone wurde noch nicht gedreht (0°) */
        for ( y = 0; y < SchablonenSize; y++)
        {
                for ( x = 0; x < SchablonenSize; x++)
                {
                        if (SchablonenLayout[y][x] == 'O')   // ausgeschnitten ?
                        {
                                if( bVerschl) // verschlüsseln
                                {
                                        VerschlNachricht[y][x] = Nachricht[ c++];
                                        if ( c >= len) break;
                                }
                                else          // entschlüsseln
                                {
                                        Nachricht[c++] = VerschlNachricht[y][x];
                                }
                        }
                }
                if ( c >= len && bVerschl) break;
        }

        /* 2.Richtung: Die Schablone wurde um 90° im Uhrzeigersinn gedreht */
        for ( x = 0; x < SchablonenSize; x++)  // 90°
        {
                for ( y = SchablonenSize - 1; y >= 0; y--)
                {
                        if (SchablonenLayout[y][x] == 'O')   // ausgeschnitten ?
                        {
                                if( bVerschl) // verschlüsseln
                                {
                                        VerschlNachricht[x][SchablonenSize - 1 - y] = Nachricht[ c++];
                                        if ( c >= len) break;
                                }

                                else          // entschlüsseln
```

```
                                        {
                                                Nachricht[ c++] = VerschlNachricht[x][SchablonenSize - 1 - y];
                                        }
                                }
                        }
                        if ( c >= len && bVerschl) break;
                }

                /* 3.Richtung: Die Schablone wurde um 180° im Uhrzeigersinn gedreht */
                for ( y = SchablonenSize - 1; y >= 0; y--) // 180°
                {
                        for ( x = SchablonenSize - 1; x >= 0; x--)
                        {
                                if (SchablonenLayout[y][x] == 'O')   // ausgeschnitten ?
                                {
                                        if( bVerschl) // verschlüsseln
                                        {
                                                VerschlNachricht[ SchablonenSize - 1 - y][SchablonenSize - 1 - x] = Nachricht[ c++];
                                                if ( c >= len) break;
                                        }
                                        else          // entschlüsseln
                                        {
                                                Nachricht[ c++] = VerschlNachricht[ SchablonenSize - 1 - y][SchablonenSize - 1 - x];
                                        }
                                }
                        }
                        if ( c >= len && bVerschl) break;
                }

                /* 4. Richtung: Die Schablon wurde um 270° im Uhrzeigersinn gedreht */
                for ( x = SchablonenSize - 1; x >= 0; x--) // 270°
                {
                        for ( y = 0; y < SchablonenSize; y++)
                        {
                                if ( SchablonenLayout[y][x] == 'O')  // ausgeschnitten ?
                                {
                                        if( bVerschl) // verschlüsseln
                                        {
                                                VerschlNachricht[SchablonenSize - 1 - x][y] = Nachricht[ c++];
                                                if ( c >= len) break;
                                        }
                                        else          // entschlüsseln
                                        {
                                                Nachricht[ c++] = VerschlNachricht[SchablonenSize - 1 - x][y];
                                        }
                                }
                        }
                        if ( c >= len && bVerschl) break;
                }
                if( bVerschl)  // Wenn verschlüsselt wurde, die verschl. Nachricht ausgeben
                {
                        printf( "\n\n\rDie verschlüsselte Nachricht lautet:\n\r");
                        for( c = 0; c < SchablonenSize; c++) printf( "%s\n\r", VerschlNachricht[ c]);
                }
                else          // Wenn entschlüsselt wurde, die entschlüsselte Nachricht ausgeben
                {
                        printf( "\n\n\rDie entschlüsselte Nachricht lautet:\n\r");
                        puts( Nachricht);
                }
}

int main( void)
{
        Init();
        switch( Menu()) // je nach Menüauswahl ver- oder entschlüsseln
        {
                case 1: Verschluesseln( true);     break;
                case 2: Verschluesseln( false);  break;
                default: printf( "Ungültige Eingabe !");
        }
        LoescheSchablone();
        return 0;
}
```

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 $t_1 + t_2 + \ldots + t_n$ 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, $t_1 = t_2 = \ldots = t_7 = 45$ 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.

Lösung von Aufgabe 2: Partyvorbereitungen

1. 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.

1.1 Die Strategie

Wenn im folgenden von großen und kleinen Aufgaben die Rede ist, so sind damit Aufgaben gemeint, die mehr oder weniger Zeit in Anspruch nehmen. Um eine möglichst effiziente Verteilung der Aufgaben und gleichzeitig einen schnellen Programmablauf zu gewährleisten, verwendet mein Programm folgende Strategie:
Es wird mit einem Helfer begonnen, dem immer die größte der noch nicht verteilten Aufgaben, für die seine Zeit noch ausreicht, zugewiesen wird. Lassen sich ihm keine weiteren Aufgaben zuteilen, da seine restliche zur Verfügung stehende Zeit kürzer ist als die Zeit, die für die kürzeste Aufgabe benötigt würde, so werden die verbleibenden Aufgaben einem weiteren Helfer nach dem selben Schema zugeteilt usw..

1.2 Beweis, daß nicht mehr als doppelt so viele Helfer wie nötig eingespannt werden

Geht man nach oben vorgestellter Strategie vor, so ist sichergestellt, daß jeder Helfer, außer dem letzten, mindestens die Hälfte der ihm zur Verfügung stehenden Zeit arbeitet. Falls ihm als erstes eine Aufgabe zugewiesen wurde, die mindestens die Hälfte seiner Zeit in Anspruch nimmt, ist obige Behauptung sowieso erfüllt. Ist dies nicht der Fall so kann jede weitere Aufgabe, die ihm zugeteilt wird, maximal so groß wie die erste sein, d.h. es können ihm in jedem Fall so lange Aufgaben zugewiesen werden, bis mehr als die Hälfte seiner Zeit verbraucht ist, außer es gibt keine weiteren Aufgaben und in diesem Fall wäre er der letzte, was oben bereits ausgeschlossen wurde. Betrachtet man nun nur die beiden letzten Helfer, so lassen sich folgende Schlüsse ziehen. Da der letzte nur die Aufgaben zugewiesen bekommt, die beim vorletzten nicht mehr untergebracht werden konnten, müssen sie zusammen mehr Zeit investieren als einer von ihnen alleine hätte aufbringen können. Wenn man die gesamte von den beiden letzten Helfern gearbeitete Zeit durch zwei teilt, um den Durchschnitt zu berechnen, zeigt sich, daß die beiden letzten Helfer im Mittel auch mindestens die Hälfte, der ihnen zur Verfügung stehenden Zeit arbeiteten müssen. Damit ist bewiesen, daß alle Helfer im Durchschnitt mindestens die Hälfte ihrer Zeit, im Normalfall sogar wesentlich mehr, mit Arbeit verbringen. Daraus folgt, daß Katrin auch bei optimaler Verteilung aller Aufgaben nicht mit der Hälfte der Helfer auskommen würde, da die Zeit, die alle Aufgaben zusammen in Anspruch nehmen würden, bereits die Zeit, die die halbe Anzahl der Helfer aufbringen könnte, übersteigen würde. Falls der Sonderfall auftritt, daß so wenig Arbeit vorhanden ist, daß alle Aufgaben von einem Helfer erledigt werden können, so läßt sich zwar der oben stehende Beweis nicht verwenden, allerdings ist dann klar, daß Katrin nicht mit der Hälfte an Helfern ausgekommen wäre, da sie nicht weniger als eine Person um Hilfe bitten kann.

2. Schreibe ein Programm, das Deine Strategie realisiert.

2.1 Lösungsidee

Die Lösungsidee entspricht in erster Linie der Strategie, die in 1.1 dargestellt wurde. Die programmtechnische Umsetzung ist in der Programm-Dokumentation näher ausgeführt.

2.2 Programm-Dokumentation

Die zu Beginn definierten Konstanten bestimmen das Verhalten des Zufallsgenerators bei der Erzeugung der Zeiten für die einzelnen Aufgaben. Im einzelnen bestimmt *ZUFMAX,* wie viele Aufgaben es insgesamt zu erledigen gibt, in diesem Beispiel also 10000. Durch *ZUFZAHL* wird festgelegt, wie lang die einzelnen Aufgaben maximal dauern dürfen, in diesem Fall 20 min. *ZUFZEIT* definiert schließlich noch die Zeit, die jeder Helfer aufbringen kann. Diese Einstellungen spielen aber nur eine Rolle, falls man mit dem Zufallsgenerator arbeitet.
Als nächstes werden nun einige globalen Variablen deklariert. Sie besitzen folgende Funktion:

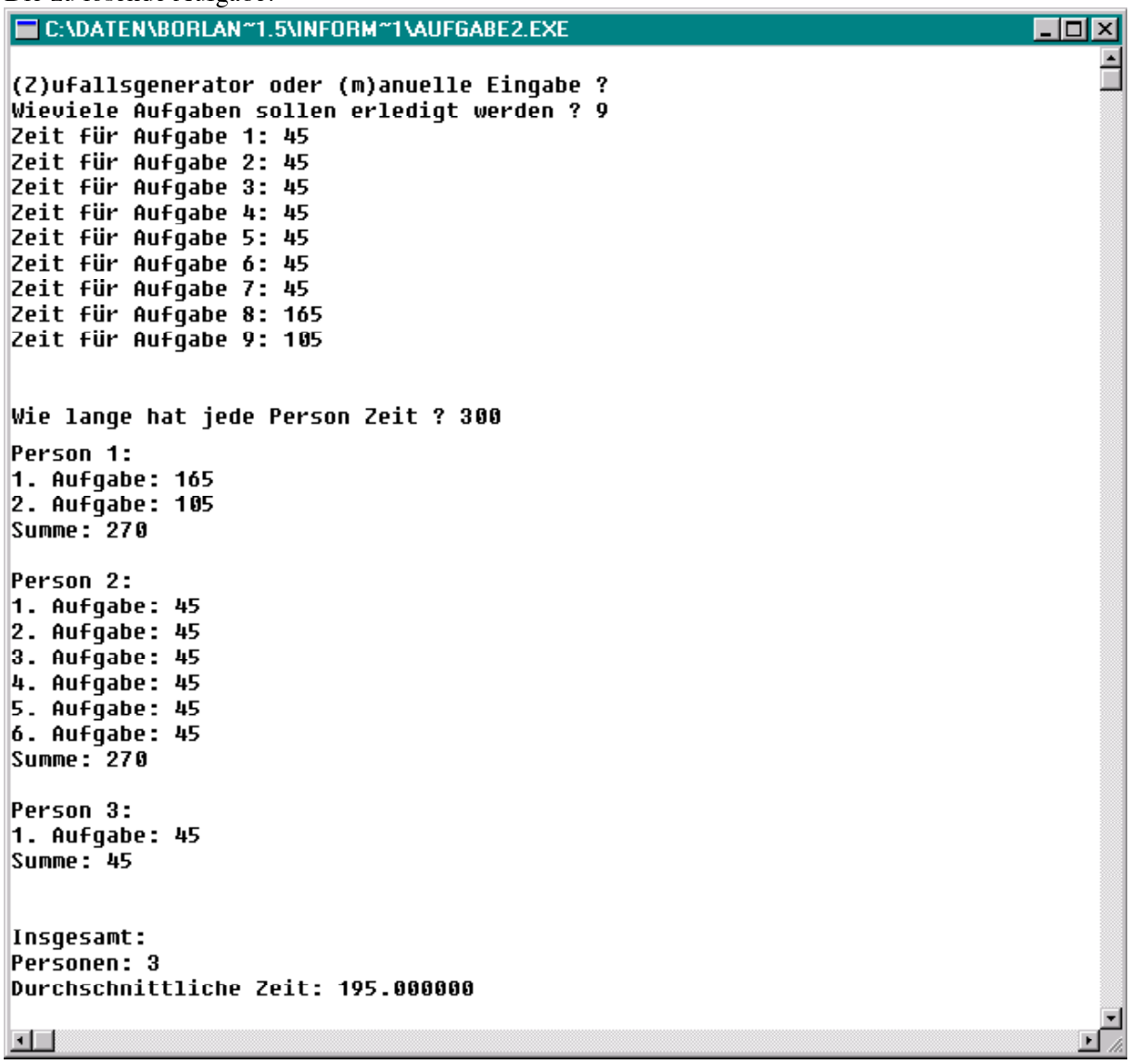
AufgabenAnzahl	legt die Gesamtzahl der zu erledigenden Aufgaben fest
pAufgaben	zeigt auf einen dynamisch reservierten Array von *int*, der alle Zeiten, die für jede einzelne Aufgaben gebraucht werden enthält
Zeit	legt fest, wie lange jeder Helfer Zeit hat, um Aufgaben zu erledigen
bZufallsgenerator	bestimmt ob der Zufallsgenerator verwendet wird (1) oder nicht (0); dies spielt nur für die Form der Ausgabe des Programms eine Rolle

Die Funktion *LeseAufgaben()* überprüft zunächst ob der Zufallsgenerator verwendet werden soll. Entscheidet sich der Benutzer durch Eingabe eines kleinen oder großen 'Z' für den Zufallsgenerator, so wird zunächst *bZufallsgenerator* auf 1 gesetzt. Anschließend wird *AufgabenAnzahl* auf *ZUFMAX* gesetzt, da *ZUFMAX,* wie oben beschrieben, die Gesamtanzahl der zu erledigenden Aufgaben enthält. Ebenso wird *Zeit* auf *ZUFZEIT* gesetzt. Nachdem für *pAufgaben* ein Integer-Array der Größe *AufgabenAnzahl* mit Hilfe von *new* reserviert wurde, wird nun jedem Speicherplatz in diesem Array, der die Zeit für eine einzelne Aufgabe enthält, eine zufällige Zeit aus dem Bereich von 1 bis *ZUFZAHL* zugewiesen. Entscheidet sich der Benutzer hingegen durch die Eingabe eines kleinen oder großen 'M' für die manuelle Eingabe, so wird *bZufallsgenerator* auf 0 gesetzt, da der Zufallsgenerator nicht benutzt wird. Anschließend wird die Zahl der zu erledigenden Aufgaben in *AufgabenAnzahl* eingelesen. Nun wird wiederum ein Array der Größe *AufgabenAnzahl* eingerichtet, auf das *pAufgaben* zeigt. In der nachfolgenden *for*-Schleife wird der Benutzer aufgefordert, die Zeit für jede einzelne Aufgabe einzugeben. Die eingegebenen Zeiten werden in dem Array *pAufgaben* gespeichert. Zum Abschluß muß noch eingegeben werden, wie lange jeder Helfer Zeit hat. Dieser Wert wird in *Zeit* abgelegt.
Die Funktion *sort_function(const void *a, const void *b)* ist eine Hilfsroutine, die von der Bibliotheksfunktion *qsort* zum Sortieren der Zeiten für die einzelnen Aufgaben in absteigender Reihenfolge im Array *pAufgaben* benötigt wird. Um eine absteigende Sortierung (d.h. die größte Zeit befindet sich nach der Sortierung in *pAufgaben[0]*) zu erreichen, gibt *sort_function* für $a < b$ die Zahl 1, für $a > b$ die Zahl -1 und für $a = b$ die Zahl 0 zurück. Beim späteren Sortieren repräsentieren *a* und *b* dabei zwei Zeiten aus *pAufgaben*. Auf Grund dieser Rückgabewerte ist es der Funktion *qsort* möglich, ein beliebiges Array unter Verwendung des QuickSort-Algorithmus zu sortieren.
Die Funktion *VerteileAufgaben()* bildet das Kernstück des Programms, da sie die eigentliche Verteilung der Aufgaben auf eine möglichst geringe Anzahl von Helfern übernimmt. Zuerst werden die einzelnen Aufgaben, ihrem Zeitbedarf nach, absteigend mit *qsort* sortiert. Dann wird überprüft, ob die längste Aufgabe (*pAufgaben*[0]) mehr Zeit in Anspruch nimmt, als ein einzelner Helfer aufbringen kann. Ist dies der Fall, kann die Aufgabe natürlich von niemandem erledigt werden, deshalb wird die Funktion nach Ausgabe einer Fehlermeldung verlassen. Die nun folgende *while*-Schleife wird solange durchlaufen, bis alle Aufgaben einem Helfer zugeteilt wurden. Sie endet erst, wenn alle Aufgaben verteilt und deshalb alle Zeiten auf 0 gesetzt wurden, denn nur dann findet sich auch, nach der Sortierung, in *pAufgaben[0]* die Zeit 0. Jeder Durchlauf dieser Schleife entspricht einem neuen Helfer, deshalb wird die Variable *nPersons*, die die Anzahl der benötigten Helfer enthält, jedesmal um eins erhöht. Die in der *while*-Schleife enthaltene *for*-Schleife durchläuft immer alle noch nicht verteilten Aufgaben in absteigender Reihenfolge nach ihrem Zeitbedarf und prüft dabei für jede Aufgabe, ob die verbleibende Zeit des Helfers für diese Aufgabe noch ausreicht. Trifft dies zu, so wird die entsprechende Zeit der Aufgabe ausgegeben, die Zeit, die dieser Helfer bereits arbeitet (im Programm: *nVerbrauchteZeit*) um die Zeit dieser

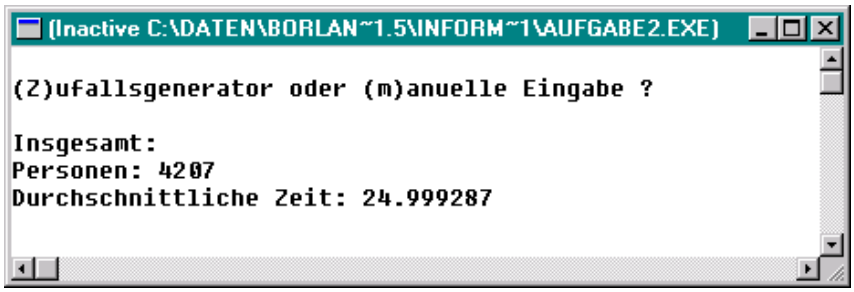
Aufgabe erhöht und die Zeit der Aufgabe im Array *pAufgaben* auf 0 gesetzt, damit sie beim nächsten Durchlauf, also beim nächsten Helfer, nicht mehr berücksichtigt wird. Die *for*-Schleife wird durch *break* vorzeitig verlassen, sobald das Programm auf die erste Null in *pAufgaben* stößt, da dies aufgrund der Sortierung bedeutet, daß nur noch weitere Nullen, d.h. keine Aufgaben mehr folgen. Am Ende der *for*-Schleife, also wenn einem Helfer keine weiteren Aufgaben mehr zugewiesen werden können, wird noch *nVerbrauchteZeit*, die Zeit, die der Helfer insgesamt arbeiten muß, ausgegeben. Außerdem wird *ZeitGesamt*, die Zeit die für alle Aufgaben zusammen benötigt wird, um *nVerbrauchteZeit* erhöht. Schließlich müssen die Aufgaben ihrem Zeitbedarf nach mit Hilfe von *qsort* wieder neu sortiert werden. Am Ende der Funktion *VerteileAufgaben()* stehen dann in *nPersons* die Gesamtzahl der benötigten Helfer und in *ZeitGesamt* die Zeit, die insgesamt gearbeitet wird, zur Verfügung. Aus diesen beiden Werten läßt sich leicht durch Division die durchschnittliche Arbeitszeit pro Helfer ausrechnen. Diese Informationen werden hier noch ausgegeben. Die Ausgaben, die sich auf einen einzelnen Helfer beziehen, werden in Abhängigkeit von *bZufallsgenerator* unterdrückt, wenn der Zufallsgenerator benutzt wird, da die Ausgabe ansonsten zu umfangreich würde.
Das Hauptprogramm schließlich überprüft durch den Rückgabewert von *LeseAufgaben(),* ob bei dem Einlesen der Aufgaben Fehler aufgetreten sind. Ist dies nicht der Fall wird die Funktion *VerteileAufgaben()* aufgerufen. Am Ende wird der Speicher, der von *pAufgaben* belegt wird, mit *delete* wieder freigegeben.

2.3 Programmablauf-Protokoll

Die zu lösende Aufgabe:



Mit dem Zufallsgenerator erzeugte Ausgabe:
(10000 Aufgaben, $t_i \in [1;20]$, Zeit die pro Person zur Verfügung steht: 25)



2.4 Programm-Text

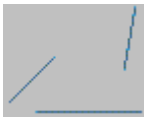
```
// Informatik-Wettbewerb
// Aufgabe 2
// (c) Michael Wack 1997
// 128 lines

#include <stdio.h>     // benötigte include-Dateien
#include <stdlib.h>
#include <conio.h>
#include <mem.h>

// Die hier definierten Konstanten kommen nur zur Geltung wenn der Zufallsgenerator benutzt wird.
#define     ZUFMAX  10000                       // Anzahl der Zeiten, die vom Zufalssgenerator erzeugt werden
#define     ZUFZAHL 20                          // Zeiten für Aufgaben werden von 1 bis zu diesem Wert erzeugt
#define     ZUFZEIT 25                          // Zeit die pro Person zur Verfügung steht

int         AufgabenAnzahl,                     // Die Anzahl der zu erledigenden Aufgaben
            *pAufgaben,                         // Zeiger auf ein Array mit den Zeiten der einzelnen Aufgaben
            Zeit,                               // Zeit die pro Person zur Verfügung steht
            bZufallsgenerator;                  // legt fest ob der Zufallsgenerator benutzt werden soll

int         LeseAufgaben();                     // Liest die Aufgaben ein oder erzeugt sie durch Zufallsgenerator
int         sort_function( const void *a, const void *b); // Hilfsfunktion zum Sortieren
void        VerteileAufgaben();                 // Nimmt die eigentliche Verteilung der Aufgaben vor


int LeseAufgaben()
{
            int res, c;
            char ch;
            printf( "\n(Z)ufallsgenerator oder (m)anuelle Eingabe ?");
            ch = getch();
            if( ch == 'Z' || ch == 'z')             // Zufallsgenerator verwenden
            {
                        bZufallsgenerator = 1;      // Zufallsgenerator wird verwendet
                        randomize();                // Zufallsgenerator initialisieren
                        AufgabenAnzahl = ZUFMAX;
                        Zeit = ZUFZEIT;
                        pAufgaben = new int[ AufgabenAnzahl];   // Speicher für Int-Array zur Aufnahme der Zeiten
                                                                // für die einzelnen Aufgaben reservieren
                        for( c = 0; c < AufgabenAnzahl; c++)
                        {
                                    pAufgaben[ c] = random( ZUFZAHL) + 1;  // Jeder Aufgabe wird eine zufällige Zeit zugewiesen
                        }
            }
            else if( ch == 'M' || ch == 'm')        // manuelle Eingabe
            {
                        bZufallsgenerator = 0;      // Zufallsgenerator wird nicht verwendet
                        printf( "\nWieviele Aufgaben sollen erledigt werden ? ");
                        res = scanf( "%d", &AufgabenAnzahl);
                        if ( res == 0 || res == EOF) return 1;          // Falls ein Fehler aufgetreten ist -> 1 zurückgeben
                        pAufgaben = new int[ AufgabenAnzahl];   // Speicher für Int-Array zur Aufnahme der Zeiten
                                                                // für die einzelnen Aufgaben reservieren
                        for( c = 0; c < AufgabenAnzahl; c++)
                        {
                                    printf( "Zeit für Aufgabe %d: ", c+1);      // Für jede zu erledigende Aufgabe
                                    scanf( "%d", &pAufgaben[ c]);               // die Zeit einlesen
                        }

                        printf( "\n\nWie lange hat jeder Helfer Zeit ? ");
                        scanf( "%d", &Zeit);        // Die Zeit die pro Helfer zur Verfügung steht einlesen
            }
            else
            {
                        printf( "\nUngültige Eingabe !");          // Fehler -> 1 zurückgeben
                        return 1;
            }
            return 0;
}


int sort_function( const void *a, const void *b)  // Hifsfunktion um mit qsort die Zeiten absteigend zu sortieren
```

```
{
	if (*(int *)a > *(int *)b) return -1;
	if (*(int *)a < *(int *)b) return  1;
	return 0;
}

void VerteileAufgaben()				// Erledigt das eigentliche Aufteilen der Aufgaben auf möglichst wenig Helfer
{
	int	nVerbrauchteZeit,		// Zeit die der aktuellen Helfer bereits arbeitet
		nPerson = 0,			// Anzahl der Helfer
		c;				// Zählvariable, die zum Durchlaufen aller Aufgaben dient
	long	ZeitGesamt = 0;			// Zeit die alle Aufgaben zusammen in Anspruch nehmen würden

	qsort( pAufgaben, AufgabenAnzahl, sizeof( pAufgaben[ 0]), sort_function); // Aufgaben der Zeit nach absteigend sortiern
	if( pAufgaben[ 0] > Zeit) // Die am längsten dauernde Aufgabe ist länger als ein einzelner Helfer Zeit aufbringen kann -> Fehler
	{
		printf( "\n\nDie Zeit einer Aufgabe überschreitet die zur Verfügung stehende Zeit !");
		return;
	}

	while( pAufgaben[ 0] != 0) // solange es noch Aufgaben zu verteilen gibt
	{
		nVerbrauchteZeit = 0;		// die Zeit die dieser Helfer bereits arbeitet ist hier natürlich noch 0
		nPerson++;			// die Anzahl der Helfer um eins erhöhen
		if( !bZufallsgenerator) printf( "\nPerson %d:", nPerson);
		for( c = 0; c < AufgabenAnzahl; c++)	// In dieser Schleife werden alle Aufgaben durchlaufen
		{					// und falls möglich einem Helfer zugewiesen
			// falls pAufgaben[ c] = 0 ist, heißt das, da die Aufgaben absteigend sortiert sind, daß keine
			// weiteren Aufgaben mehr folgen, deshalb wird die Schleife hier verlassen
			if( pAufgaben[ c] == 0) break;
			if( nVerbrauchteZeit + pAufgaben[ c] <= Zeit)		// falls der Helfer die Aufgabe pAufgaben[ c]
			{							// noch erledigen kann
				// wird sie hier ausgegeben falls der Zufallsgenerator nicht aktiv ist
				if( !bZufallsgenerator) printf( "\n%d. Aufgabe: %d", c+1, pAufgaben[ c]);
				// und ihre Zeit zur verbrauchten Zeit des Helfers hinzu addiert
				nVerbrauchteZeit += pAufgaben[ c];
				pAufgaben[ c] = 0;	// die Zeit für diese Aufgabe wird auf 0 gesetzt, und verschiebt sich somit
			}					// bei der nächsten Sortierung an das Ende der Liste und wird somit nicht
		}						// mehr berücksichtigt
		// Nachdem dem Helfer keine weiteren Aufgaben zugewiesen werden konnten wird hier die Zeit ausgegeben,
		// die der Helfer arbeiten muß
		if( !bZufallsgenerator) printf( "\nSumme: %d\n", nVerbrauchteZeit);
		ZeitGesamt += nVerbrauchteZeit; // die Zeit die dieser Helfer arbeitet zur insgesamt verbrauchten Zeit addieren
		// hier werden die Aufgaben wieder absteigend sortiert
		qsort( pAufgaben, AufgabenAnzahl, sizeof( pAufgaben[ 0]), sort_function);
	}
	// Zum Abschluss wird hier die durchschnittliche Arbeitszeit pro Helfer ausgegeben
	printf( "\n\nInsgesamt:\nPersonen: %d\nDurchschnittliche Zeit: %f\n", nPerson, (float)ZeitGesamt/(float)nPerson);
}

int main( void)		// Hauptprogramm
{
	if( LeseAufgaben()) return 1;		// wenn die Aufgaben fehlerfrei eingelesen wurden
	VerteileAufgaben();			// dann die Aufgaben an die Helfer verteilen
	if( pAufgaben) delete [] pAufgaben;	// und zum Schluß den Speicher wieder frei geben
	return 0;
}
```

Aufgabe 3

Rekursive Besen

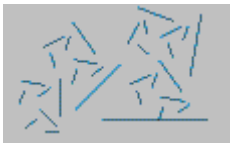
Heiner Huschelmutz, CAD-Spezialist, ist verzweifelt. Seine Frau war eine Woche auf Dienstreise und kommt morgen zurück. Huschelmutz möchte ihr die Wohnung in blitzblankem Zustand präsentieren; im Moment gleicht sie noch einem Schlachtfeld. Er will einen Besen holen, kann aber keinen finden; es ist nach 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:



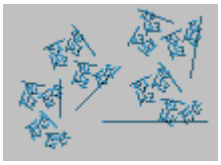
Heiner Huschelmutzs Sicht verschwimmt, und er sieht immer mehr Borsten, erst so:

dann so:

dann so:



Noch etwas später hat er folgendes Bild vor Augen:

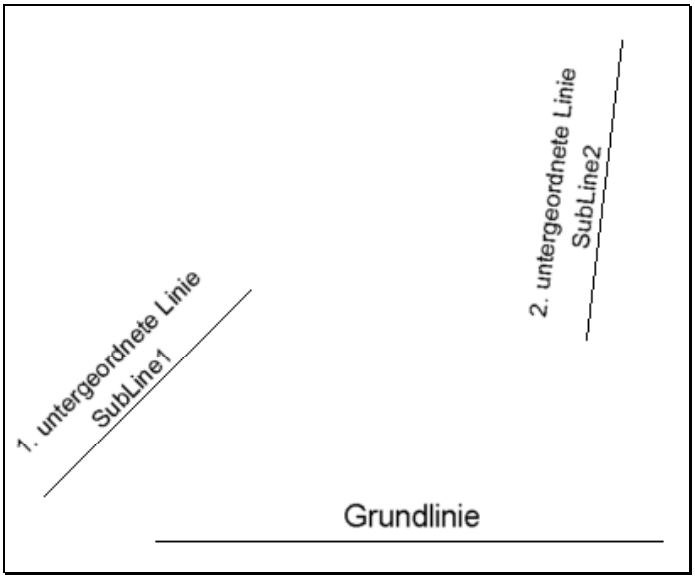


Anfänglich ist Huschelmutz verwirrt, doch bald erkennt er ein Prinzip hinter dem Spiel, das seine Augen mit ihm treiben.

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.

Lösung von Aufgabe 3: Rekursive Besen

1. Nach welchem Prinzip ist Heiner Huschelmutzs Vision aus der ursprünglichen Borstenanordnung entstanden?



In den folgenden Ausführungen beziehe ich mich auf die Bezeichnungen, die in nebenstehender Grundfigur enthalten sind. *SubLine1* und *SubLine2* sind dabei die Bezeichnungen, die im Programm verwendet werden.
Heiner Huschelmutzs Vision entsteht dadurch, daß jede der untergeordneten Linien zu einer neuen Grundlinie wird. An jeder neuen Grundlinie entsteht wieder eine mit der ursprünglichen Grundfigur identische Figur, nur daß diese entsprechend der neuen Grundlinie eine andere Größe besitzt und gedreht ist. Durch die ständige Wiederholung dieses Vorgangs entsteht ein Fraktal, da sich das Gesamtbild beliebig oft in sich selbst wiederfinden läßt. Man spricht hier von Selbstähnlichkeit.

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.

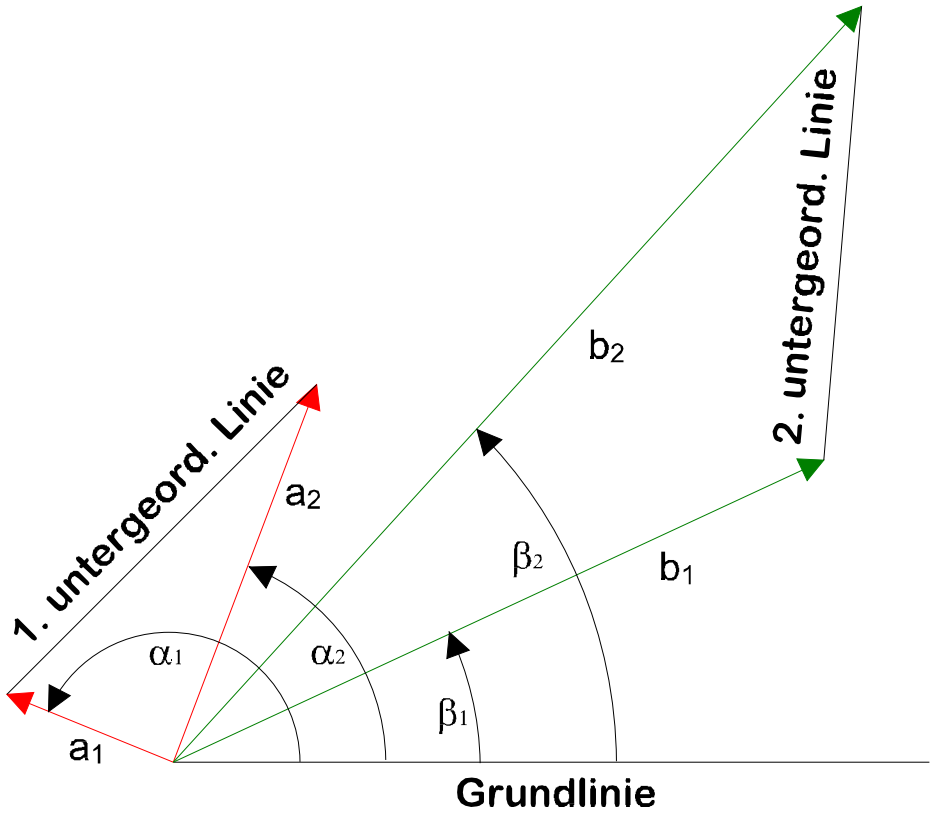
2.1 Lösungsidee

Wie im Aufgabentitel schon verraten wurde, läßt sich eine solche Aufgabe sehr einfach rekursiv lösen. Man benötigt dazu lediglich eine Funktion, die basierend auf einer gegebenen Grundlinie in der Lage ist, die beiden untergeordneten Linien passend dazu zu zeichnen. Anschließend muß sich diese Funktion wiederum selbst aufrufen, aber diesmal mit den gerade gezeichneten untergeordneten Linien als neue Grundlinien. Damit das Programm nicht endlos so weitermacht, muß durch eine Variable, die jedesmal um eins erhöht wird, wenn die Funktion aufgerufen, und um eins erniedrigt wird, wenn die Funktion verlassen wird, sichergestellt werden, daß sich die Funktion nicht mehr selbst aufruft, wenn diese Variable einen definierten Grenzwert erreicht hat. Dieser Grenzwert bestimmt die Komplexität der entstehenden Zeichnung.

2.2 Programm-Dokumentation

Um nun eine Funktion, wie die oben vorgestellte, implementieren zu können, muß man zunächst die beiden untergeordneten Linien in relativen Koordinaten zur Grundlinie speichern. Ich wählte dafür folgendes Verfahren: Die beiden Endpunkte einer untergeordnete Linie werden jeweils durch ihre relative Neigung zur Grundlinie und ihrer Entfernung zum linken Endpunkt der Grundlinie, die als Bruchteil der Grundlinienlänge angegeben wird, festgelegt.

Graphisch stellt sich dieser Sachverhalt folgendermaßen dar:

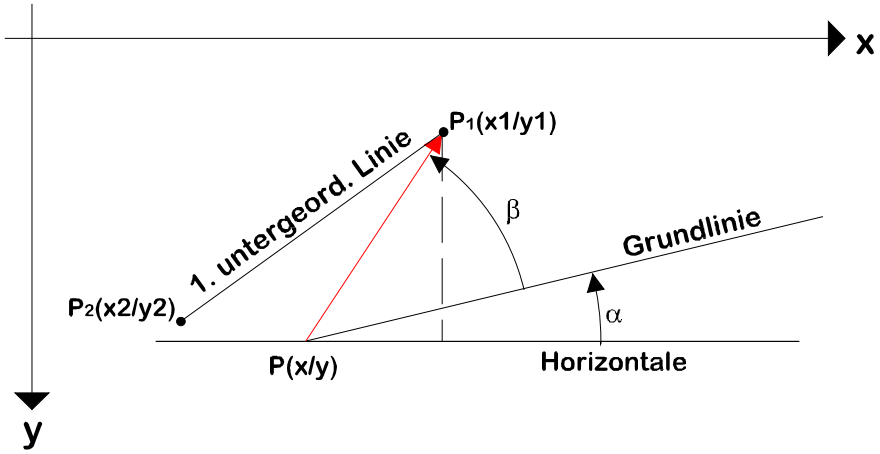


In dieser Abbildung wird die erste untergeordnete Linie durch die Entfernung a_1 und den zugehörigen Winkel α_1 sowie a_2 und α_2 festgelegt. Für die zweite untergeordnete Linie gilt entsprechendes, nur eben mit b und β. Im einzelnen ergibt sich folgende Aufstellung:

Bez.	Entsprechung im Programm	Startwerte für die Beispielaufgabe
a_1	*SubLine1.length1*	0.235
α_1	*SubLine1.angle1*	158° im Bogenmaß
a_2	*SubLine1.length2*	0.529
α_2	*SubLine1.angle2*	69° im Bogenmaß
b_1	*SubLine2.length1*	0.941
β_1	*SubLine2.angle1*	25° im Bogenmaß
b_2	*SubLine2.length2*	1.353
β_2	*SubLine2.angle2*	47° im Bogenmaß

Die Winkel sind grundsätzlich im Bogenmaß angegeben, da die trigonometrischen Funktionen in C++ dieses Format erwarten. Die Längen sind als Bruchteile der Grundlinie angegeben, d.h. bei einer Grundlinienlänge von 10 bedeutet ein Wert von 0.235 für a_1, daß die Strecke a_1 die absolute Länge 2.35 hat.
Die Funktion *DrawSubLines(double angle, double length, int x, int y)* übernimmt das Zeichnen der untergeordneten Linien zu einer gegebenen Grundlinie. Der Parameter *angle* legt die Neigung der Grundlinie gegenüber der Horizontalen fest. Dieser Winkel muß im Bogenmaß angegeben werden. *length* bestimmt die absolute Länge der Grundlinie in Pixeln, *x* und *y* legen schließlich die absoluten Anfangskoordinaten der Grundlinie fest. Aus diesen Angaben und den weiter oben erläuterten relativen Positionen der untergeordneten Linien ist es der Funktion möglich, mit Hilfe von trigonometrischen Überlegungen die beiden untergeordneten Linien zu zeichnen. Dazu werden zunächst für jede untergeordnete Linie die beiden absoluten Endpunkte, die durch *x1*, *y1* bzw. *x2*, *y2* festgelegt sind, bestimmt. Wie die Funktion die absoluten Koordinaten errechnet, soll

hier exemplarisch an x1 und y1 der ersten untergeordneten Linie gezeigt werden. Alle anderen Punktkoordinaten werden analog ermittelt.



Wie aus obiger Zeichnung ersichtlich ist, erhält man die absoluten Koordinaten des Punktes P_1 folgendermaßen:

$$x_1 = \cos(\alpha + \beta) * [PP_1] + x$$
$$y_1 = y - \sin(\alpha + \beta) * [PP_1]$$

Dabei gilt:

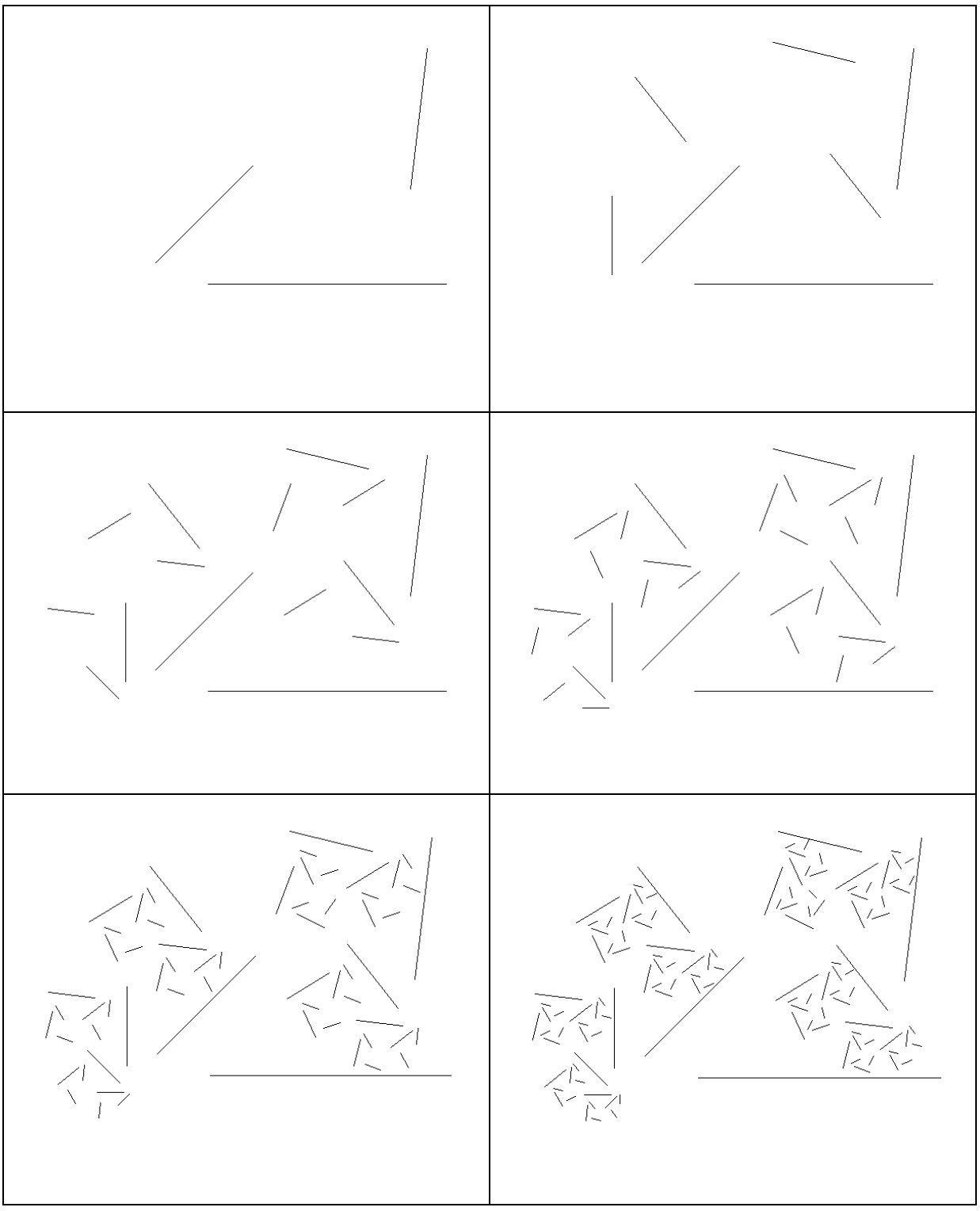
$[PP_1]$ =	SubLine1.length1 * length	Längenfaktor der untergeordneten Linie * Grundlinienlänge
α =	angle	Winkel zwischen Grundlinie und Horizontalen
β =	SubLine1.angle1	Winkel zwischen Grundlinie und $[PP_1]$

Hieraus ergeben sich die im Programm verwendeten Gleichungen:
*x1 = cos(SubLine1.angle1 + angle) * SubLine1.length1 * length + x;*
*y1 = y - sin(SubLine1.angle1 + angle) * SubLine1.length1 * length;*

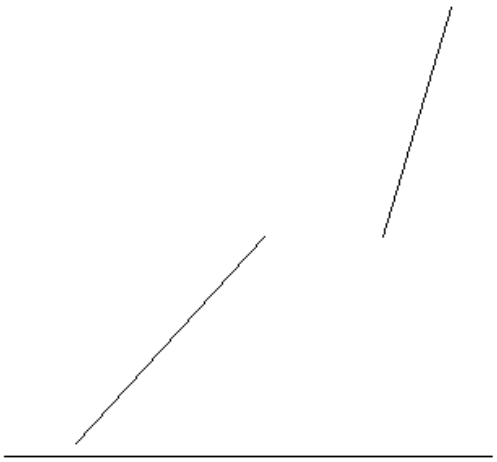
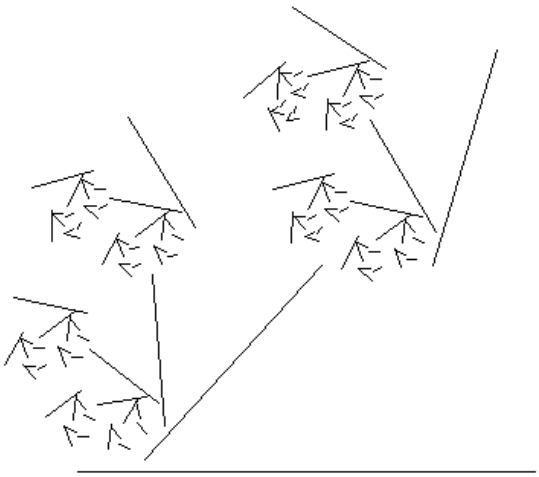
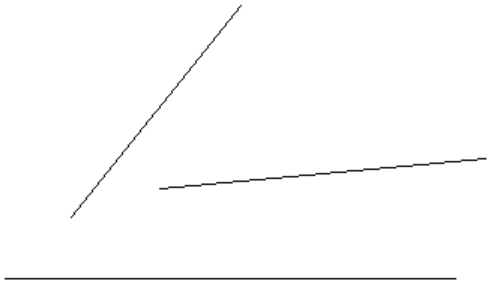
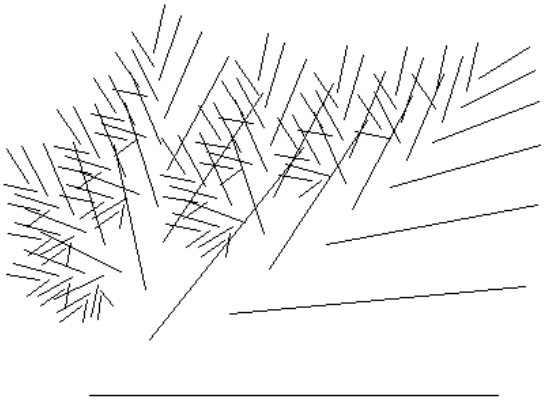
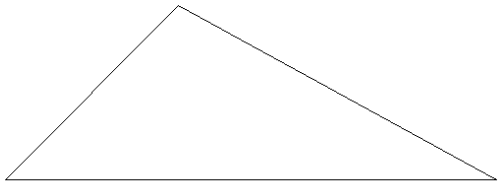
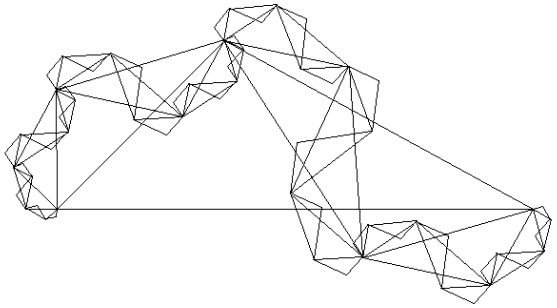
Nachdem alle Koordinaten auf diese Weise ermittelt wurden, wird die Linie gezeichnet. Anschließend wird die Funktion *DrawSubLines* rekursiv aufgerufen. Die Neigung der neuen Grundlinie wird mit Hilfe des atan(ΔY / ΔX) errechnet, die Länge der Linie wird mit Hilfe des Pythagoras ($a^2 + b^2 = c^2$) ermittelt. *x1* und *x2* werden zu den neuen Ausgangskoordinaten der Grundlinie. Mit der zweiten untergeordneten Linie wird auf die gleiche Art verfahren. Die globale Variable *Level* wird jeweils zu Beginn der Funktion *DrawSubLines* um eins erhöht und am Ende um eins erniedrigt. Sie kann deshalb dazu verwendet werden, den Rekursionsprozess bei einer bestimmten Stufe zu stoppen. Im Programm wurde dies so umgesetzt, daß die Funktion *DrawSubLines* sofort zu Beginn wieder verlassen wird, falls eine bestimmte Rekursionsstufe erreicht wurde. Diese Grenze wird durch die Konstante *MaxLevel* bestimmt.
Die Funktion *Init()* übernimmt den Wechsel in den Grafikmodus mit Hilfe von *initgraph* und initialisiert die Variable *Level* mit einem Startwert von 0.
Das Hauptprogramm beginnt mit dem Aufruf der Routine Init() und zeichnet die erste Grundlinie, da diese von der Funktion *DrawSubLines* nicht gezeichnet wird. Anschließend wird durch den Aufruf von *DrawSubLines* der Rekursionsprozess in Gang gesetzt. Als Parameter werden, die zu Beginn des Programms definierten Konstanten StartX, StartY und GroundLength verwendet. Sie legen die absolute Position der Grundlinie sowie ihre Länge fest und dienen damit zum Skalieren und Positionieren des gesamten Bildes. Wurde das ganze Bild gezeichnet und die vom Hauptprogramm aufgerufene Funktion *DrawSubLines* wieder verlassen, wartet das Programm auf einen Tastendruck, um dann den Grafikmodus zu beenden.

2.3 Pogrammablauf-Protokoll

Hier folgen nun Bilder der ersten sechs Stufen der zu lösenden Beispielaufgabe:



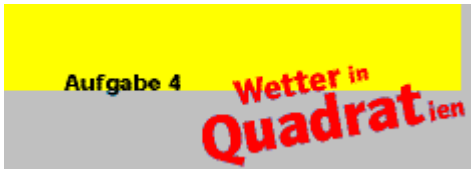
Und hier nochmals drei Bilder mit anderen Grundfiguren:

Grundfigur:	Mit sechs Rekursionsebenen:
	
	
	

2.4 Programm-Text

```
// Informatik-Wettbewerb
// Aufgabe 3
// Rekursive Besen
// (c) Michael Wack 1997
// 102 lines

#define MaxLevel        6       // Bis zu dieser Stufe wird das Bild gezeichnet
#define StartX          250     // X-Koordinate des Anfangspunktes der Grundlinie
#define StartY          250     // Y-Koordinate des Anfangspunktes der Grundlinie
#define GroundLength    200     // Länge der Grundlinie

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>
#include <math.h>

// Definition der beiden untergeordneten Linien:
// angle bestimmt den Winkel zwischen der Grundlinie und dem jeweiligen Endpunkt der untergeordneten Linie im Bogenmaß.
// length bestimmt die Entfernung von dem linken Ende der Grundline bis zum
// jeweiligen Endpunkt der untergeordneten Linie in Bruchteilen der Grundlinie.

const struct _SubLine       // legt die Lage einer untergeordneten Linie relativ zu einer
{                           // Grundlinie fest.
    double angle1, angle2, length1, length2;
} SubLine1 = {              (double)M_PI/(double)180*(double)158,    // 158° im Bogenmaß
                (double)M_PI/(double)180*(double)69,     // 69° im Bogenmaß
                0.235, 0.529                             // 23.5%, 52.9%
                },
 SubLine2 = {               (double)M_PI/(double)180*(double)25,     // 25° im Bogenmaß
                (double)M_PI/(double)180*(double)47,     // 47° im Bogenmaß
                0.941, 1.353                             // 94.1%, 135.3%
                };

int Level;      // enthält immer die aktuelle Rekursionstiefe

void Init();    // Initialisiert die Grafik und setzt Standardwerte


// zeichnet zu einer gegebenen Grundlinie die entsprechenden untergeord. Linien
void DrawSubLines( double angle, double length, int x, int y);


void Init()
{
    // Grafik initialisieren
    int gd, gm, res;
    gd = DETECT;
    initgraph( &gd, &gm, "e:\\progra~1\\bc45\\bgi");

    res = graphresult();
```

```
    if (res != grOk)
    {
            printf( "Grafik konnte nicht initialisiert werden !\nFehler: %s", grapherrormsg( res));
            getch();
            exit( 1);
    }

    Level = 0;
}

void DrawSubLines( double angle, double length, int x, int y)
{
    int x1, x2, y1, y2;

    if (Level >= MaxLevel) return; // falls wir in der tiefsten Rekursionsebene sind -> abbrechen

    Level++;            // Rekursionsebene um eins erhöhen

    // 1. untergeordnete Linie
    x1 = cos( SubLine1.angle1 + angle) * SubLine1.length1 * length + x;
    y1 = y - sin( SubLine1.angle1 + angle) * SubLine1.length1 * length;
    x2 = cos( SubLine1.angle2 + angle) * SubLine1.length2 * length + x;
    y2 = y - sin( SubLine1.angle2 + angle) * SubLine1.length2 * length;
    line( x1, y1, x2, y2);

    // untergeordnete Linie als neue Grundlinie benutzen
    DrawSubLines( atan2((double)(y1 - y2), (double)(x2 - x1)),
                    sqrt((double)(x2-x1)*(double)(x2-x1) + (double)(y1-y2)*(double)(y1-y2)), x1, y1);

    // 2. untergeordnete Linie
    x1 = cos( SubLine2.angle1 + angle) * SubLine2.length1 * length + x;
    y1 = y - sin( SubLine2.angle1 + angle) * SubLine2.length1 * length;
    x2 = cos( SubLine2.angle2 + angle) * SubLine2.length2 * length + x;
    y2 = y - sin( SubLine2.angle2 + angle) * SubLine2.length2 * length;
    line( x1, y1, x2, y2);

    // untergeordnete Linie als neue Grundlinie benutzen
    DrawSubLines( atan2((double)(y1 - y2), (double)(x2 - x1)),
                    sqrt((double)(x2-x1)*(double)(x2-x1) + (double)(y1-y2)*(double)(y1-y2)), x1, y1);

    Level--;    // Rekursionseben um eins verringern
}


int main( void)
{
    Init();             // Grafik initialisieren und global Variablen mit Startwerten laden
    line( StartX, StartY, StartX + GroundLength, StartY);  // Grundlinie zeichnen
    DrawSubLines( 0, GroundLength, StartX, StartY);             // Rekursionsprozess initieren
    getch();            // Warten bis eine Taste gedrückt wird
    closegraph();       // Grafikmodus verlassen
    return 0;
}
```

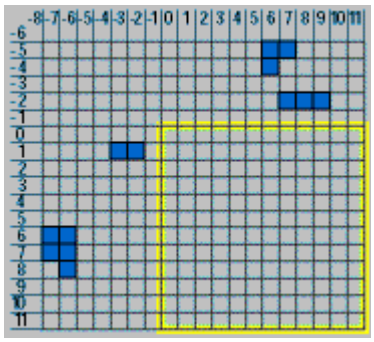


Quadratien ist ein quadratisches Gebiet aus quadratischen Feldern. Das Feld in der Nordwest-Ecke hat die Zeilennummer 0 und die Spaltennummer 0.

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 von Aufgabe 4: Wetter in Quadratien

1. Schreibe ein Programm, welches folgendes leistet:

- **Einlesen der Größe von Quadratien (Anzahl Zeilen bzw. Spalten)**
- **Einlesen einer Wolkenvorhersage**
- **Ausgabe, wo in Quadratien wie oft Regen fällt**

1.1 Lösungsidee

Um eine Regenvorhersage erstellen zu können, muß die Bewegung der Wolken über Quadratien simuliert werden. Dazu müssen die Wolken, die in Nord-Süd-Richtung ziehen, bei jedem Takt um zwei Felder Richtung Süden und die West-Ost-Wolken um drei Felder nach Osten bewegt werden. So läßt sich nach jedem Takt überprüfen, ob sich auf einem Feld zwei Wolken befinden und somit mit Regen zu rechnen ist.
Die naheliegende und wahrscheinlich einfachere Möglichkeit die Felder von Quadratien in einem zweidimensionalen Array zu speichern und in jedem Feld festzuhalten, ob sich über ihm eine NS- und/oder eine WE-Wolke befindet und außerdem, wie oft es an dieser Stelle bereits geregnet hat, stößt schnell an Grenzen. Vor allem gibt es Probleme, wenn man Wolken eingeben will, die noch sehr weit von Quadratien entfernt sind, denn dann müßte man das Array auch entsprechend vergrößern. Dies ist allerdings nur begrenzt möglich, denn bereits bei einer Arraygröße von zehntausend mal zehntausend Feldern und unter der Annahme, daß man alle Informationen zu einem Feld in einem einzigen Byte unterbringen könne, würde man bereits 100 MB (!) Speicher brauchen.
Deshalb entschied ich mich dafür, die Wolken einer Richtung in jeweils einer doppelt verketteten Liste zu speichern. Diese Listen werden in Zukunft auch Wolkenlisten genannt. Bei jedem Takt wird nun entsprechend der Richtung, eine Koordinate einer jeden Wolke erhöht und anschließend nach Wolken gesucht, die sich über dem selben Feld befinden. Lassen sich solche Wolken finden, werden die Koordinaten dieses Feldes in eine weitere Liste eingetragen. Diese Liste wird im Weiteren auch als Felderliste bezeichnet. Gibt es bereits ein Feld mit den selben Koordinaten in dieser Liste, so wird nur die Anzahl der Regenfälle in diesem Feld um eins erhöht. Die beiden Wolken, aus denen der Regen entstand, werden anschließend aus den entsprechenden Wolkenlisten herausgelöscht.

1.2 Programmdokumentation

Die im Abschnitt 1.1 erwähnten Listen wurden folgendermaßen implementiert: Die am Anfang des Programm-Texts deklarierten Strukturen (*struct*) *_RainField* und *_Cloud* stellen einzelne Elemente einer Wolken- bzw. der Felderliste dar. Sie enthalten jeweils einen Zeiger *pNext* auf das nächste Element. Im Fall von *_Cloud* existiert auch noch ein Zeiger *pPrev*, welcher auf die vorhergehende Wolke zeigt. Deshalb handelt es sich hier um eine doppelt und nicht nur eine einfach verkettete Liste. Dieser Zusammenhang ist wichtig, weil er das Entfernen von einzelnen Wolken aus der Liste erheblich vereinfacht. Da aus der Felderliste nichts gelöscht werden muß, reicht hier eine einfach verkettete Liste.
Die globalen Zeigervariablen, die von diesen Typen abgeleitet wurden, zeigen jeweils auf das erste Element einer Liste. So zeigt *pNSClouds* auf das erste Element der Liste der Nord-Süd-Wolken und *pWEClouds* auf das erste Element der West-Ost-Wolken-Liste; *pRainFields* zeigt dementsprechend auf das erste Element der Felderliste.
Die Funktion *GetQuadratienSize()* dient zum Einlesen der Seitenlänge von Quadratien in die globale Variable *QuadratienSize*. Falls bei der Eingabe ein Fehler auftritt, wird eins zurückgegeben andernfalls null.
GetClouds() übernimmt das Einlesen der Positionen der Wolken und fügt sie, entsprechend ihrer Anfangsposition, in die zugehörigen Listen ein. So werden Wolken, die sich im Norden von Quadratien befinden, automatisch in die NS-Liste und Wolken im Westen in die WE-Liste eingefügt. Wird die Position einer Wolke, die sich bereits in Quadratien befindet, eingegeben, so wird der Benutzer zur Eingabe einer Zugrichtung aufgefordert, da es in diesem Fall dem Programm nicht möglich ist, diese eindeutig zu bestimmen. Die Eingabe von Wolken, die sich im NW, O oder S befinden, wird, nach

Ausgabe einer entsprechenden Fehlermeldung, ignoriert, da sie nie über Quadratien hinwegziehen werden. Das Einlesen der Wolken-Koordinaten läßt sich durch die Eingabe eines beliebigen Buchstabens beenden. Im Detail läuft das Einlesen einer einzelnen Wolke in folgenden Schritten ab. Zuerst werden die beiden Koordinaten, die durch ein Leerzeichen getrennt sein müssen, mit *gets* in den String *str* eingelesen. Anschließend werden sie mittels *sscanf* in die beiden lokalen Variablen *x* und *y* übertragen. Falls dies nicht möglich ist, da der Benutzer zum Beispiel einen Buchstaben statt der Koordinaten eingegeben hat, wird die *while*-Schleife hier durch *break* verlassen. *x* beinhaltet nun die WE-Koordinate und *y* die NS-Koordinate. Zuerst wird überprüft, ob sich die eingegebene Wolke innerhalb von Quadratien befindet. Dies ist der Fall, wenn folgendes gilt: *(0 <= x < QuadratienSize und 0 <= y < QuadratienSize).* Da sich die Zug-richtung nicht vorhersagen läßt, wird der Benutzer zur Eingabe der Himmelsrichtung, in die sich die Wolke bewegen soll, aufgefordert. Andernfalls ist es möglich die Zugrichtung der Wolke aufgrund folgender Bedingungen zu ermitteln:

NS-Wolke: 0 <= x < QuadratienSize und y < 0
WE-Wolke: 0 <= y < QuadratienSize und x < 0

Ist keine der obigen Bedingungen erfüllt, so erfolgt die Ausgabe einer entsprechenden Fehlermeldung und die Funktion fährt mit dem Einlesen der nächsten Wolke fort. Konnte die Zugrichtung der Wolke hingegen eindeutig bestimmt werden, so wird sie in die, ihrer Zugrichtung entsprechenden, Liste zu Beginn eingefügt. Dies bedeutet konkret, daß zunächst eine neue Wolke mittels *new* erzeugt wird und *pNewCloud* ein Zeiger auf diese Wolke zugewiesen wird. Dann werden die Koordinaten der Wolke auf die vorher eingelesenen x- und y-Werte gesetzt und die Zeigervariablen mit null initialisiert. Danach wird überprüft, ob es überhaupt schon ein erstes Element in der Liste gibt. Hierzu wird festgestellt, ob *pNSClouds* bzw. *pWEClouds* den Wert null aufweist. Ist dies der Fall, so wird *pNSClouds* bzw. *pWEClouds* der Wert von *pNewCloud* zugewiesen, so daß *pNSClouds* bzw. *pWEClouds* nun auf die neue Wolke zeigt. Gibt es bereits andere Wolken in der jeweiligen Liste, so wird *pNSClouds* bzw. *pWEClouds* ebenfalls mit *pNewCloud* gleichgesetzt, aber erst nachdem *pNewCloud->pNext* auf die ehemalige erste Wolke zeigt. Damit die Integrität der Liste erhalten bleibt, muß der Zeiger *pPrev* der ehemalig ersten Wolke auf die neu hinzugefügte Wolke zeigen. Damit ist das Einfügen der neuen Wolke zu Beginn der passenden Liste abgeschlossen und das Programm beginnt mit dem Einlesen der Koordinaten der nächsten Wolke.

Die Funktion *FreeAll()* stellt fest, von welchen Listen Speicher belegt wurde und gibt ihn anschließend frei. Hierzu werden alle Elemente einer jeden Liste durchlaufen und dabei mittels *delete* gelöscht.

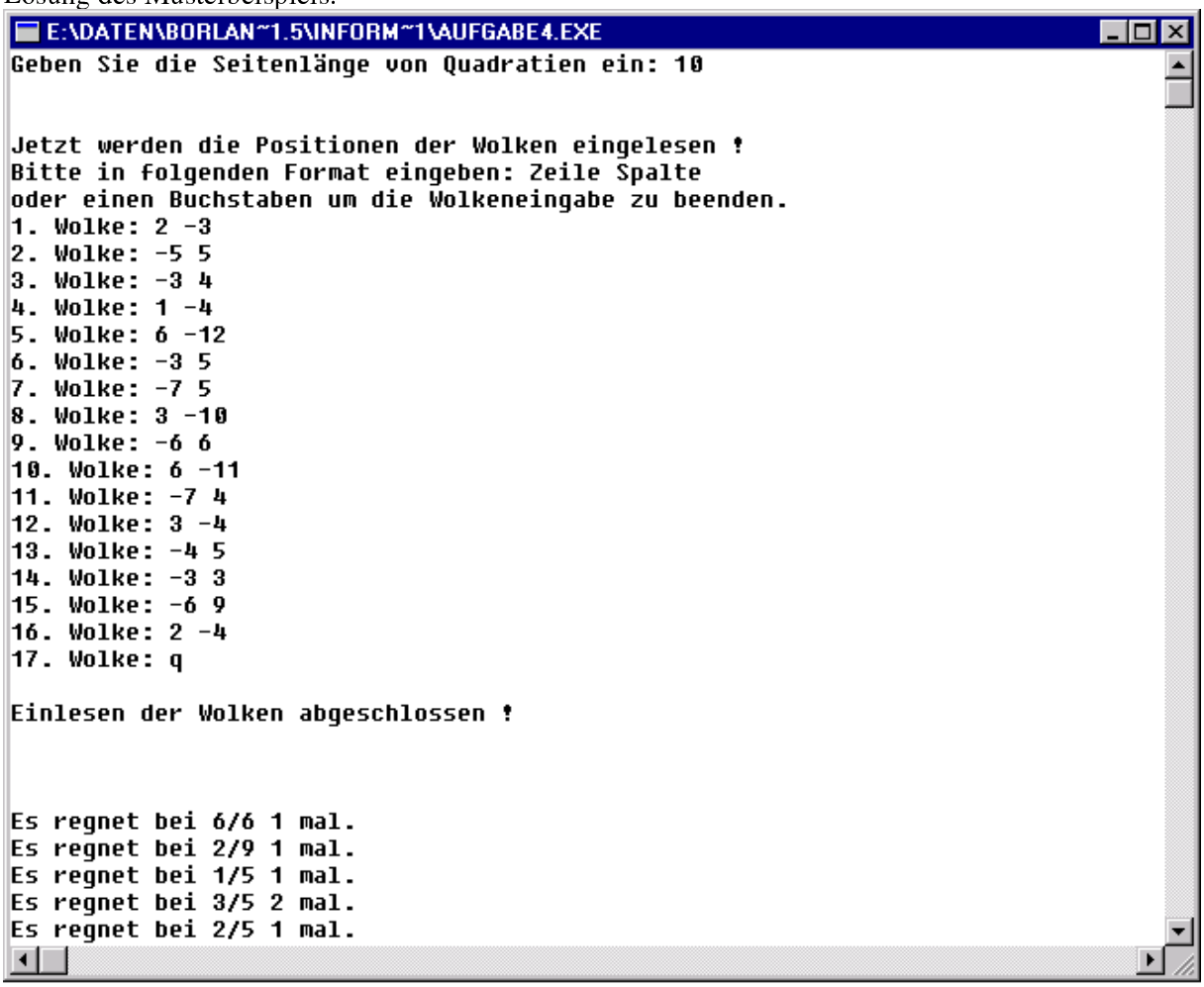
MakeWeatherForecast() übernimmt die Auswertung der eingelesenen Wolken und gibt an-schließend die Regenvorhersage für Quadratien aus. Um dies zu erreichen, werden bei jedem Takt zuerst die NS-Wolken um jeweils zwei Felder nach Süden weiter gerückt. Hierzu wird jedes Element der NS-Wolkenliste durchlaufen und jeweils die Y-Koordinate um zwei erhöht. Mit den WE-Wolken wird in gleicher Weise verfahren, allerdings wird hier, nach dem Ändern der X-Koordinate, jede WE-Wolke mit allen NS-Wolken verglichen, um festzustellen, ob sich zwei Wolken zur gleichen Zeit über dem selben Feld befinden. Ist dies der Fall, so regnet es an dieser Stelle und die Koordinaten des Feldes werden zur Felderliste hinzugefügt. Gibt es bereits ein Feld mit diesen Koordinaten in der Felderliste, so wird dessen Variable *Count*, die die Anzahl der Regenfälle auf diesem Feld enthält, um eins erhöht. Anschließend werden die beiden beteiligten Wolken aus den Wolkenlisten gelöscht, da sie sich auflösen. Die Variablen *nNSCloudsLeft* und *nWECloudsLeft* enthalten, nach der Bewegung aller Wolken innerhalb eines Taktes, die Anzahl der Wolken, die nach Abschluß des nächsten Taktes noch über Quadratien hinwegziehen könnten. Durch diese Werte wird sichergestellt, daß das Programm beendet wird, sobald keine weiteren Wolken mehr existieren, die sich begegnen könnten. Um am Ende der Funktion *MakeWeatherForecast()* die Regenvorhersage erstellen zu können, wird die Felderliste durchlaufen und die Koordinaten *X* und *Y* eines jeden Feldes sowie *Count*, also wie oft es in diesem Feld insgesamt geregnet hat, ausgegeben .

Das Hauptprogramm liest zuerst mittels *GetQuadratienSize()* die Seitenlänge von Quadratien ein. Anschließend erfaßt *GetClouds()* die Wolken. Treten hierbei Fehler auf, so wird mit *FreeAll()* der belegte Speicher wieder freigegeben und das Programm beendet. Ansonsten erstellt die Funktion *MakeWeatherForecast()* die Regenvorhersage und zeigt diese an. Am Ende wird auch in diesem Fall der Speicher wieder freigegeben.

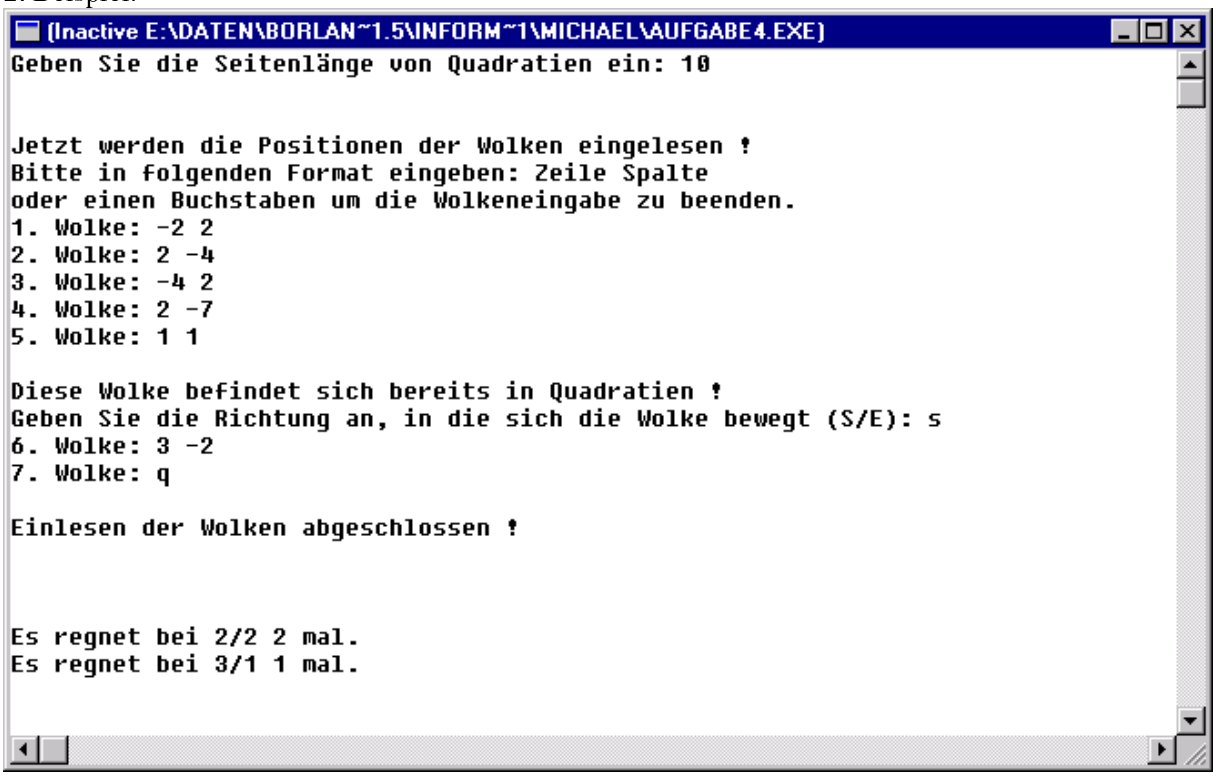
Bei der Eingabe sollte man darauf achten, niemals mehrere Wolken mit den gleichen Koordinaten einzugeben, da im Fall eines Regenschauers nur eine der gleichen Wolken gelöscht würde und die anderen deshalb ganz normal weiterziehen würden. Auf eine Überprüfung durch das Programm habe ich zu Gunsten der Übersichtlichkeit des Quelltexts verzichtet.

1.3 Programmablauf-Protokoll

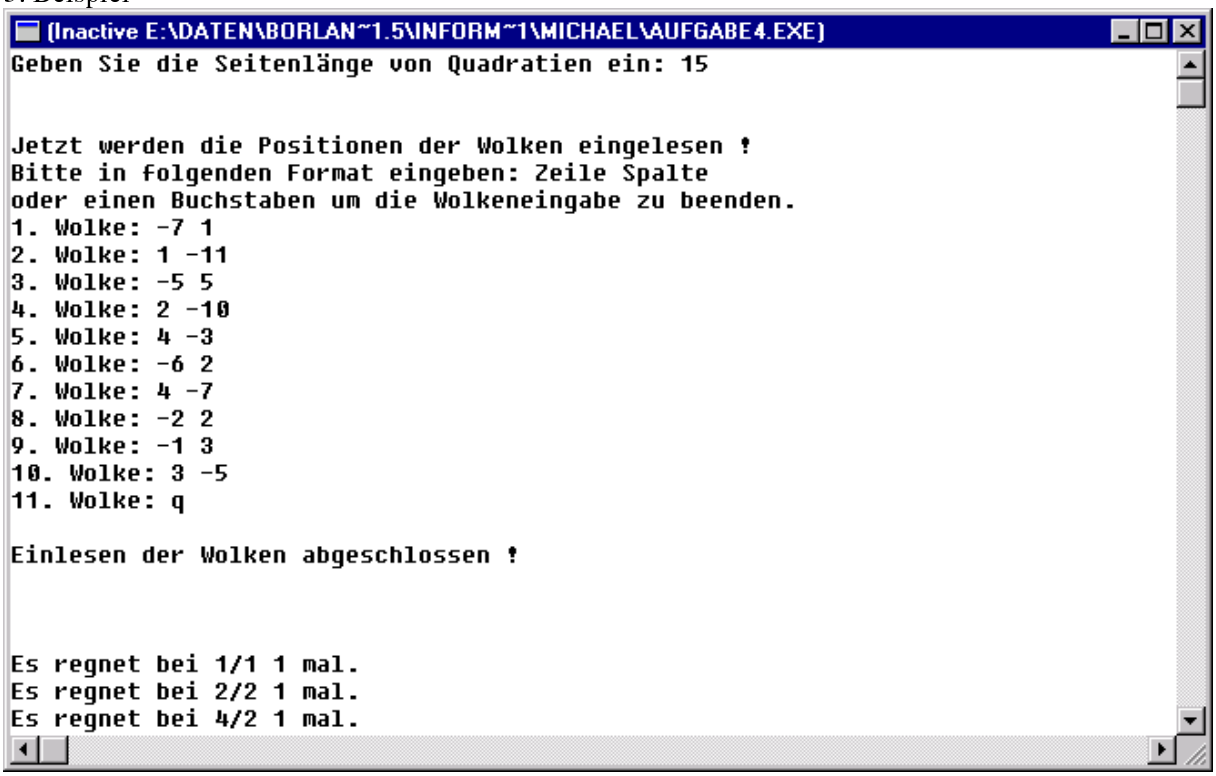
Lösung des Musterbeispiels:



2. Beispiel:



3. Beispiel


(Inactive E:\DATEN\BORLAN~1.5\INFORM~1\MICHAEL\AUFGABE4.EXE)
Geben Sie die Seitenlänge von Quadratien ein: 15
Jetzt werden die Positionen der Wolken eingelesen !
Bitte in folgenden Format eingeben: Zeile Spalte
oder einen Buchstaben um die Wolkeneingabe zu beenden.
1. Wolke: -7 1
2. Wolke: 1 -11
3. Wolke: -5 5
4. Wolke: 2 -10
5. Wolke: 4 -3
6. Wolke: -6 2
7. Wolke: 4 -7
8. Wolke: -2 2
9. Wolke: -1 3
10. Wolke: 3 -5
11. Wolke: q
Einlesen der Wolken abgeschlossen !
Es regnet bei 1/1 1 mal.
Es regnet bei 2/2 1 mal.
Es regnet bei 4/2 1 mal.

1.4 Programm-Text

```
// Informatik-Wettbewerb
// Aufgabe 4
// Wetter in Quadratien
// (c) Michael Wack 1997
// 247 lines

#include <stdio.h>
#include <conio.h>

struct _RainField                       // Element der einfach verketteten Liste der Felder, in denen bereits Regen fiel
{
        int X, Y, Count;                // X und Y Koordinate des Feldes sowie die Anzahl der Regenfälle
        _RainField *pNext;              // Zeiger auf das nächste Feld in der Liste
} *pRainFields;                         // das erste Element der Liste

struct _Cloud                           // Element einer doppelt verketteten Liste von Wolken
{
        int X, Y;                       // Aktuelle Position der Wolke
        _Cloud *pNext, *pPrev;          // Zeiger auf die vorhergehende und die nächste Wolke in der Liste
} *pNSClouds, *pWEClouds;               // Zeiger auf die ersten Elemente der Listen für die NS-Wolken und die WE-Wolken

int     QuadratienSize;                 // Seitenlänge von Quadratien einlesen

int     GetQuadratienSize();            // Liest die Seitenlänge von Quadratien ein
int     GetClouds();                    // Liest die Positionen der Wolken ein
void    FreeAll();                      // Gibt allen belgten Speicher wieder frei
void    MakeWeatherForeCast();          // Erstellt einen Wetterbericht

int GetQuadratienSize()    // Seitenlänge von Quadratien einlesen
{
        char str[ 10];
        int res;
        printf( "Geben Sie die Seitenlänge von Quadratien ein: ");
        gets( str);                 // Seitenlänge als String einlesen
        res = sscanf( str, "%d", &QuadratienSize);  // String in int umwandeln und in QuadratienSize speichern
        if (res == 0 || res == EOF || QuadratienSize <= 0)
        {
                printf( "\nFehler bei der Eingabe !");
                return 1;    // Fehler
        }
        return 0;       // Alles in Ordnung
}

int GetClouds()          // Liest die Positionen der Wolken ein
{
        int res, c, x, y;
        char str[ 10], ch;
        _Cloud *pNewCloud;

        printf( "\n\nJetzt werden die Positionen der Wolken eingelesen !");
        printf( "\nBitte in folgenden Format eingeben: Zeile Spalte");
        printf( "\noder einen Buchstaben um die Wolkeneingabe zu beenden.\n");

        c = 0; res = 1;
        while( 1)
        {
                printf( "%d. Wolke: ", ++c);
                gets( str);
                res = sscanf( str, "%d%d", &y, &x);
                if( res != 0 && res != EOF) // Falls beim Einlesen der x- und y-Koordinate keine Fehler
                {                       // aufgetreten sind, dann
                        pNewCloud = new _Cloud;                 // eine neue Wolke erstellen
                        pNewCloud->X = x; pNewCloud->Y = y;         // und ihr die eingelesenen Koordinaten zuweisen
                        pNewCloud->pPrev = 0; pNewCloud->pNext = 0;

                        ch = 0;



                        if( x >= 0 && x < QuadratienSize && y >= 0 && y < QuadratienSize) // Wolke befindet sich in Quadratien
```

```
                                {
                                        printf( "\nDiese Wolke befindet sich bereits in Quadratien !");
                                        printf( "\nGeben Sie die Richtung an, in die sich die Wolke bewegt (S/E): ");
                                        ch = getche(); // ch == s -> Wolke zieht nach Süden; ch == e -> Wokke nach Osten
                                        printf( "\n");
                                }

                                if( ((x >= 0 && x < QuadratienSize) && (y < 0)) || (ch == 'S') || (ch == 's')) // NS-Wolke
                                {
                                        if( !pNSClouds)                         // Falls es noch keine anderen Wolken gibt
                                        {
                                                pNSClouds = pNewCloud;          // muß pNSClouds auf die neue zeigen
                                        }
                                        else                                    // ansonsten wird die neue Wolke
                                        {                                       // vor allen anderen in die Liste eingefügt
                                                pNewCloud->pNext = pNSClouds;           // die nächste Wolke ist die bisher erste
                                                // die vorhergehende Wolke der nächsten Wolke ist nätürlich die neue Wolke
                                                pNewCloud->pNext->pPrev = pNewCloud;
                                                pNSClouds = pNewCloud;  // die neue Wolke wird zur ersten Wolke
                                        }
                                }
                                else if( ((x < 0) && (y >= 0 && y < QuadratienSize)) || (ch == 'E') || (ch == 'e')) // WE-Wolke
                                {
                                        if( !pWEClouds)                         // Falls es noch keine anderen Wolken gibt
                                        {
                                                pWEClouds = pNewCloud;          // muß pWEClouds auf die neue zeigen
                                        }
                                        else
                                        {
                                                pNewCloud->pNext = pWEClouds;   // die nächste Wolke ist die bisher erste
                                                // die vorhergehende Wolke der nächsten Wolke ist nätürlich die neue Wolke
                                                pNewCloud->pNext->pPrev = pNewCloud;
                                                pWEClouds = pNewCloud;  // die neue Wolke wird zur ersten Wolke
                                        }
                                }
                                else    // Befindet sich die eingegebene Wolke weder im Norden noch im Westen von Quadratien,
                                {       // dann wird sie ignoriert, da sich ihre Zugrichtung nicht bestimmen läßt
                                        printf( "\nDiese Wolke wird niemals über Quadratien hinwegziehen");
                                        printf( "\noder es wurde eine ungültige Richtung angegeben");
                                        printf( "\nund wird deshlalb ignoriert.\n");
                                        delete pNewCloud;
                                }
                        }
                        else break;// Falls keine Zahlen als Koordinaten eingegeben wurden, sondern ein Buchstabe
                }                       // so wird das Einlesen hier abgebrochen

                printf( "\nEinlesen der Wolken abgeschlossen !\n\n\n");

                if( !pNSClouds || !pWEClouds)   // Falls es überhaupt keine Wolken, oder nur Wolken einer
                {                               // Richtung gibt, dann wurden falsche Werte eingegeben
                        printf( "\n\nFehler bei der Wolkeneingabe");
                        return 1; // Fehler
                }

                return 0;
}

void FreeAll() // Löscht alle Wolken
{
                _Cloud *pCloud;
                _RainField *pRainField1, *pRainField2;
                if( pNSClouds)          // NS-Wlken löschen
                {
                        pCloud = pNSClouds;
                        while( pCloud->pNext)
                        {
                                pCloud = pCloud->pNext;
                                delete pCloud->pPrev;
                        }
                        delete pCloud; // Wolke endgültig löschen
                }


                if( pWEClouds)          // WE-Wolken löschen
                {
```

```
                pCloud = pWEClouds;
                while( pCloud->pNext)
                {
                        pCloud = pCloud->pNext;
                        delete pCloud->pPrev;
                }
                delete pCloud; // Wolke endgültig löschen
        }

        pRainField1 = pRainFields;      // Felderliste löschen
        while( pRainField1)
        {
                pRainField2 = pRainField1->pNext;
                delete pRainField1; // Feld endgültig löschen
                pRainField1 = pRainField2;
        }
}

void MakeWeatherForecast()      // Erstellt Regenvorhersage
{
        _Cloud *pNSCloud, *pWECloud, *pCloud;
        _RainField *pRainField;
        int nNSCloudsLeft, nWECloudsLeft; // Anzahl der Wolken die noch über Quadratien hinwegziehen könnten.
        int bFoundRainField;

        do   // Diese Schleife wird für jeden Takt durchlaufen, solange noch Wolken über Quadratien hinwegziehen
        {
                nNSCloudsLeft = 0; nWECloudsLeft = 0;  // Verbleibende Wolken mit 0 initialisieren
                pNSCloud = pNSClouds;
                do          // Diese Schleife wird für alle NS-Wolken durchlaufen
                {
                        pNSCloud->Y += 2;     // in Y-Richtung, also Richtung Süden um zwei Feldes weiterrücken
                        // Falls sich die Wolke auch nach dem nächsten Takt noch über Quadratien befindet,
                        // dann wird die Anzahl der verbleibenden NS-Wolken um eins erhöht
                        if( pNSCloud->Y + 2 <= QuadratienSize) nNSCloudsLeft++;
                        pNSCloud = pNSCloud->pNext;
                } while( pNSCloud); // Solange es noch weitere Wolken gibt

                pWECloud = pWEClouds;
                do          // Diese Schleife wird für alle WE-Wolken durchlaufen
                {
                        pWECloud->X += 3;     // in X-Richtung, also Richtung Osten um drei Felder weiterrücken
                        // Falls sich die Wolke auch nach dem nächsten Takt noch über Quadratien befindet,
                        // dann wird die Anzahl der verbleibenden WE-Wolken um eins erhöht
                        if( pWECloud->X + 3 <= QuadratienSize) nWECloudsLeft++;
                        pNSCloud = pNSClouds;

                        do // Diese Schleife wird für alle NS-Wolken durchlaufen
                        {
                                // WECloud wird hier mit jeder NS-Wolke verglichen um fest zu stellen ob sie sich über
                                // dem selben Feld befinden und es regenen wird.
                                if( pNSCloud->Y == pWECloud->Y && pNSCloud->X == pWECloud->X)
                                {  // Regen
                                        bFoundRainField = 0;

                                        if( pRainFields)
                                        {
                                                pRainField = pRainFields;
                                                do   // Hier wird überprüft ob es an dieser Stelle schon einmal geregnet hat
                                                {
                                                        if( pRainField->X == pNSCloud->X &&
                                                                        pRainField->Y == pNSCloud->Y)
                                                        {           // Da es auf diesem Feld bereits geregnet hat wird die
                                                                    // Anzahl der Regenfälle um eins erhöht
                                                                    pRainField->Count++;
                                                                    bFoundRainField = 1;
                                                                    break;
                                                        }
                                                        else pRainField = pRainField->pNext;
                                                } while( pRainField);
                                        }

                                        if( !bFoundRainField) // Falls es an dieser Stelle noch nicht geregnet hat,
                                        {                     // wird ein neues Feld in die Felderliste eingefügt
                                                pRainField = new _RainField;
```

```
                                                pRainField->pNext = pRainFields;
                                                pRainField->X = pNSCloud->X;
                                                pRainField->Y = pNSCloud->Y;
                                                pRainField->Count = 1;
                                                pRainFields = pRainField;
                                        }


                                        // beteiligte Wolken löschen:
                                        pCloud = pNSCloud;
                                        if( pCloud->pPrev) pCloud->pPrev->pNext = pCloud->pNext;
                                        if( pCloud->pNext) pCloud->pNext->pPrev = pCloud->pPrev;
                                        delete pCloud;

                                        pCloud = pWECloud;
                                        if( pCloud->pPrev) pCloud->pPrev->pNext = pCloud->pNext;
                                        if( pCloud->pNext) pCloud->pNext->pPrev = pCloud->pPrev;
                                        delete pCloud;
                                }
                                pNSCloud = pNSCloud->pNext;
                        } while( pNSCloud);
                        pWECloud = pWECloud->pNext;
                } while( pWECloud);
        } while( nNSCloudsLeft && nWECloudsLeft); // solange noch Wolken aus beiden Richtungen kommen

        if( !pRainFields)       printf( "\n Es regnet nie !"); // Falls es keine Felder in der Felderliste gibt

        pRainField = pRainFields;
        while( pRainField)          // Zum Schluß erfolgt hier die Ausgabe der Regenvorhersage
        {
                printf( "\nEs regnet bei %d/%d %d mal.", pRainField->Y, pRainField->X, pRainField->Count);
                pRainField = pRainField->pNext;
        }
}

int main( void)
{
        pNSClouds = 0; pWEClouds = 0; pRainFields = 0;

        if (GetQuadratienSize()) return 1; // Größe von Quadratien einlesen
        if (GetClouds())        // Wolken einlesen
        {
                FreeAll();  // Bei einem Fehler den belegten Speicher wieder freigeben
                return 1;
        }

        MakeWeatherForecast(); // Wetterbericht erstellen

        FreeAll();    // Speicher freigeben
        return 0;
}
```

Aufgabe 5
Nach der Party

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 von Aufgabe 5: Nach der Party

1. Schreibe ein Programm, das nach der Eingabe der Gäste und der Bekanntschaften im Bekanntenkreis von Katrin ermittelt, wie oft Robert von dem Fest hört.

1.1 Lösungsidee

Um ermitteln zu können, wie oft Robert von der Party bei Katrin hört, werden die Bekanntschaften bereits bei der Eingabe zu Gruppen zusammengefaßt. Dabei besteht eine Gruppe aus Personen, bei denen sicher ist, daß alle Angehörigen dieser Gruppe von der Party Kenntnis erlangen, sobald einer von ihnen etwas davon weiß.
Beispiel: Wenn Manuel Michael und Christoph kennt, ergeben sich folgende drei Möglichkeiten:

1. Manuel informiert Michael und Christoph.
2. Manuel erfährt von Michael von der Party und informiert Christoph.
3. Manuel erfährt von Christoph von der Party und erzählt Michael davon.

Dieses Beispiel zeigt deutlich, daß, sobald irgend jemand aus dieser Gruppe von der Party hört, in jedem Fall auch alle anderen davon erfahren. Diese Tatsache macht sich mein Programm zunutze, in dem es sämtliche eingegebenen Beziehungen in solche Gruppen aufteilt und danach versucht, die Gruppen, die mindestens ein Mitglied gemeinsam haben, zu einer einzigen Gruppe zusammenzufassen. Dabei werden die Partygäste auch zu einer eigenen Gruppe zusammengefaßt. Wird dieser Vorgang oft genug wiederholt, so bleiben am Ende nur noch Gruppen übrig, die nichts mehr mit einander zu tun haben. Wählt man nun die Gruppe aus, der die Partygäste angehören, und sucht hier alle Personen heraus, die Robert kennen, so weiß man, von wem Robert von der Party erfährt.

1.2 Programm-Dokumentation

Zu Beginn das Programm-Textes wird der globale Zeiger pPersons deklariert. Er zeigt auf das erste Element einer Liste, in der alle Personen verwaltet werden. Dabei ist er vom Typ _Person, der ein einzelnes Element dieser Liste darstellt. Jedes Element beinhaltet den Namen *Name* der Person, sowie Informationen darüber, ob die Person Robert kennt (*bKnowRobert*). *nGroup* legt den Index derjenigen Gruppe fest, zu welcher die Person gehört *pNext* und *pPrev* sorgen für die Verkettung mit der nächsten bzw. der vorhergehenden Person.
Die Funktion *AddPerson(char *Name, int bKnowRobert, int nGroup)* dient dazu, eine neue Person an das Ende der oben beschriebenen Personenliste anzuhängen. Dazu werden ihr die, eine Person bestimmenden, Informationen übergeben. Im Detail wird zuerst mittels *new* Speicher für eine neue Person reserviert und ihr Zeiger *pNewPerson* zugewiesen. Nun werden die übergebenen Parameter der neuen Person zugewiesen. Anschließend wird überprüft, ob es bereits Personen in der Liste gibt. Dies geschieht durch Prüfung der globalen Variablen *pPersons,* die immer auf die erste Person in der Liste zeigt. Ist sie null, so wird ihr der Wert von *pNewPerson* zugewiesen, da dies nun die erste Person ist. Andernfalls wird mit Hilfe einer *while*-Schleife die Liste bis zum letzten Element, welches sich dann in *pPerson* befindet, durchlaufen. Dann wird *pPerson->pNext* auf *pNewPerson* und *pNewPerson->pPrev* auf *pPerson* gesetzt. Damit wurde die Liste um die neue Person erweitert. Die Funktion gibt einen Zeiger auf die neu eingefügte Person zurück.
ReadGroup(char str)* extrahiert aus *str* die einzelnen Mitglieder einer Gruppe und bestimmt dabei, wer von ihnen Robert kennt. Dabei muß *str* folgendes Format besitzen: „Manuel,Michael,Christoph“ bedeutet zum Beispiel, daß Manuel Michael und Christoph kennt. In diesem Format muß auch die Eingabe erfolgen. Katrin und Robert werden nicht als Mitglieder eingelesen, da sie niemanden etwas weitererzählen. Ist Robert ein Bekannter der ersten Person, so wird das Attribut *bKnowRobert* der ersten Person auf 1 gesetzt. Befindet sich Robert an erster Stelle, so wird *bKnowRobert* aller folgenden Personen auf 1 gesetzt, da er alle und somit auch sie ihn kennen. In diesem Fall oder wenn sich Katrin an erster Stelle befindet, werden außerdem die restlichen Mitglieder jeweils einer eigenen Gruppe zu-

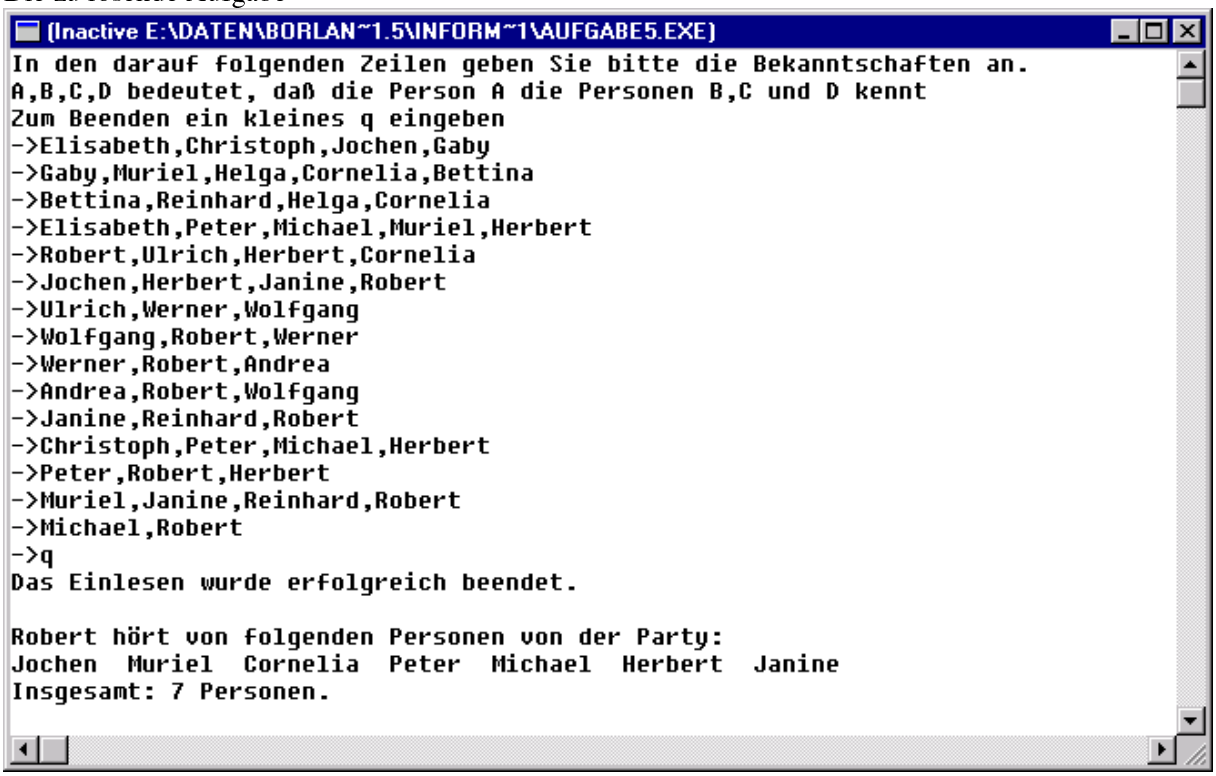
geordnet, da sie sich gegenseitig nicht kennen und Robert bzw. Katrin keine Informationen über die Party weiterleiten. Die Funktion gibt die Anzahl der Mitglieder zurück.
Mit *Input()* wird die gesamte Eingabe des Programms abgewickelt. Hierzu werden in einer Schleife so lange Gruppen mit *ReadGroup* eingelesen, bis die Eingabe eines kleinen q erfolgt. Bei der Eingabe ist zu beachten, daß in die erste Zeile die Partygäste und in die darauf folgenden Zeilen die Beziehungen zwischen den einzelnen Personen eingegeben werden müssen. Bei fehlerfreier Ausführung wird hier eine Null zurückgegeben.
Die Funktion *Output()* erledigt die Ausgabe. Hier werden, wie in der Lösungsidee bereits angesprochen, alle Personen ausgegeben, die der Gruppe 0 angehören, also von der Party gehört haben und außerdem Robert kennen (*bKnowRobert = 1*). Mit einer *while*-Schleife wird die gesamte Personenliste durchlaufen. Bei jeder Person, die den oben geforderten Kriterien entspricht, erfolgt die Ausgabe des Namens. Außerdem wird der Zähler *nPersonCount* erhöht, so daß nach dem Verlassen der Schleife die Anzahl der Personen, von denen Robert von der Party erzählt bekommt, ausgegeben werden kann.
Die Funktion *Analyze()* versucht die einzelnen Gruppen sukzessive zusammenzufassen. Dazu wird jede Person mit jeder anderen verglichen um festzustellen, ob es zweimal die selbe Person in verschiedenen Gruppen gibt. Ist dies der Fall, so werden die Gruppen dadurch zusammengefaßt, daß die *nGroup*-Variablen der Personen der einen Gruppe mit dem Index der anderen Gruppe gleichgesetzt werden. Dieser Vorgang wird jedoch nur durchgeführt, wenn nicht der Gruppenindex der Gruppe 0 verändert würde, denn die Gruppe 0 dient zur Identifizierung derjenigen Personen, die von dem Fest wissen. Wurden zwei Gruppen zusammen gefaßt, so wird die Person, die nun doppelt vorkäme, gelöscht. Falls die gelöschte Person Robert kannte, so wird diese Information auf das Duplikat dieser Person, welches nicht gelöscht wurde, übertragen. Da dieser Ablauf auch durchgeführt wird, wenn mehrmals dieselbe Person in derselben Gruppe gefunden wird, ist sichergestellt, daß Robert nicht mehrfach von der gleichen Person über die Party informiert wird.
Formal stellen sich die Abläufe in der Funktion folgendermaßen dar:

```
Wiederhole, solange noch Listen zusammen gefaßt werden
{
    Lasse pPerson1 die gesamte Liste durchlaufen
    {
        Lasse pPerson2 die gesamte Liste durchlaufen
        {
            Wenn Person1 = Person2
            {
                // Listen zusammen fassen
                Gruppe von Person1 feststellen
                Person2 löschen
                Alle Personen der Gruppe von Person2 in die von Person1 übernehmen
            }
        }
    }
}
```

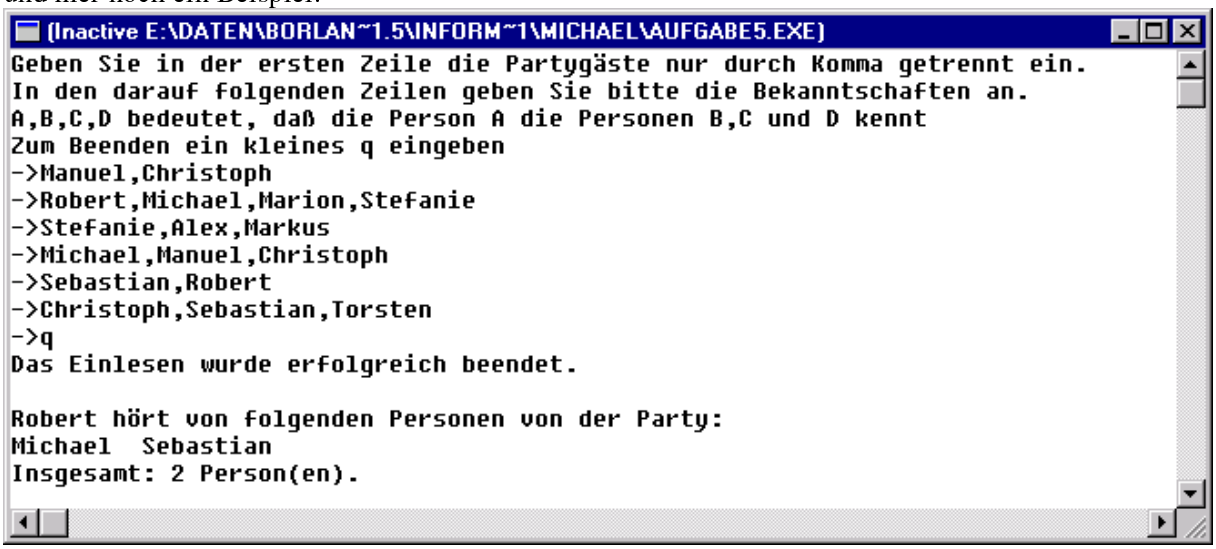
FreePersons() gibt den, von der Personenliste belegten Speicher frei, indem es die einzelnen Elemente der Liste nacheinander löscht.
Das Hauptprogramm gliedert sich in folgende Schritte:
1. Einlesen aller benötigten Informationen mit *Input()*.
2. Auswerten der einzelnen Gruppe durch *Analyze()*.
3. Ausgabe der Ergebnisse mit *Output()*.
4. Freigeben des nicht mehr benötigten Speichers mit *FreePersons()*.

1.3 Programmablauf-Protokoll

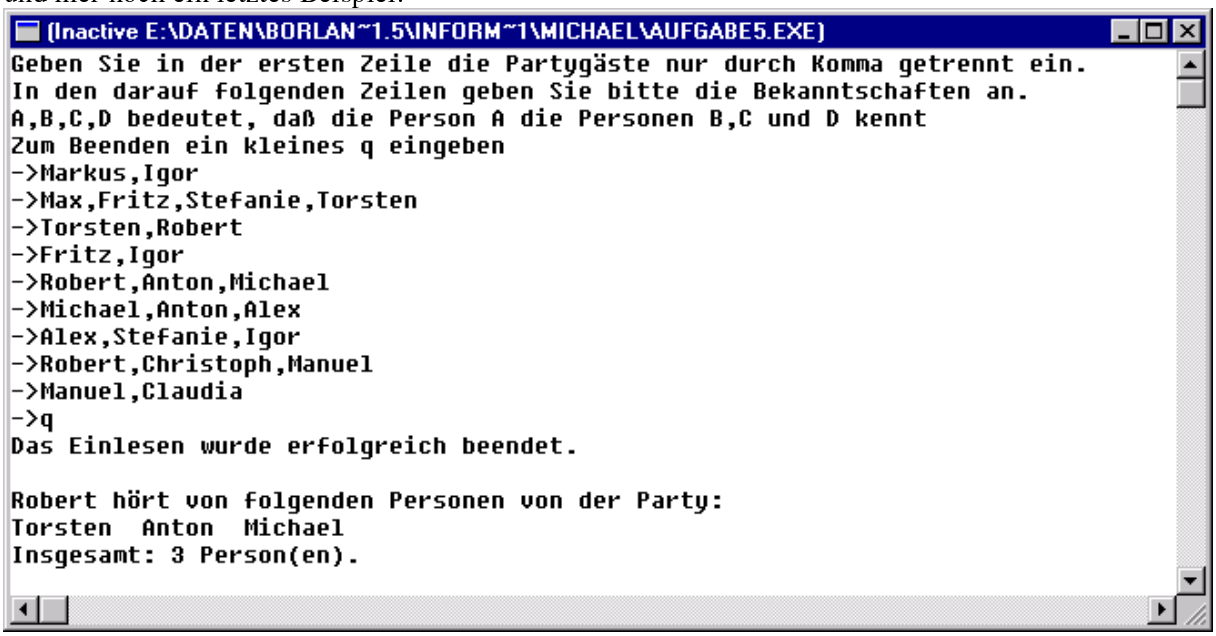
Die zu lösende Aufgabe



und hier noch ein Beispiel:



und hier noch ein letztes Beispiel:



1.4 Programm-Text

```
// Informatik-Wettbewerb
// Aufgabe 5
// Nach der Party
// (c) Michael Wack 1997
// 236 lines

#include <stdio.h>
#include <string.h>

struct _Person
{
    char        Name[ 21];      // der Name der Person
    int         bKnowRobert;    // legt fest ob diese Person Robert kennt
    int         nGroup;         // Gruppenindex
    _Person *pNext;             // Zeiger auf die nächste Person in der Liste
    _Person *pPrev;                     // Zeiger auf die vorher gehende Person in der Liste
} *pPersons;                    // Liste von Personen

int nGroup;     // globale Variable, die zur Verwaltung der Gruppen benutzt wird

_Person*    AddPerson( char* Name, int bKnowRobert, int nGroup); // Hängt eine neue Person ans Ende der Liste ein
int         ReadGroup( char* str);                  // Liest eine Gruppe von Personen ein
int         Input();                                // Liest die Beziehungen und die Partygäste ein
void        Output();                               // Gibt das Ergebnis aus
void        Analyze();                              // Wertet die Beziehungen zwischen den Personen aus
void        FreePersons();                          // Gibt den Speicher, der von der Personenliste belegt wird, frei

_Person*    AddPerson( char* Name, int bKnowRobert, int nGroup)
{
    _Person *pNewPerson, *pPerson;
    pNewPerson = new _Person;           // Speicher für eine neue Person reservieren
    strcpy( pNewPerson->Name, Name);    // Werte der neuen Person zuweisen
    pNewPerson->bKnowRobert = bKnowRobert;
    pNewPerson->nGroup = nGroup;
    pNewPerson->pNext = 0;
    pNewPerson->pPrev = 0;

    if( !pPersons) pPersons = pNewPerson;   // wenn es noch keine Personen in der Liste gibt
    else
    {
        // letzte Person suchen
        pPerson = pPersons;
        while( pPerson->pNext) pPerson = pPerson->pNext;
        // neue Person an das Ende anfügen
        pPerson->pNext = pNewPerson;            // die Person, die nach der letzten folgt, soll die neue sein
        pNewPerson->pPrev = pPerson;        // die vorhergehende Person der neuen ist die ehemalig letzte
    }
    return pNewPerson;  // Zeiger auf die neue Person zurückgeben
}
```

```
int ReadGroup( char *str)               // Liest eine Gruppe, die sich durch Komma getrennt in str befindet
{
    _Person *pFirstPerson;                      // zeigt auf die erste Person, die aus str extrahiert wurde
    int bFirstIsRobert, bFirstIsKatrin;         // legt fest ob sich Robert bzw. Katrin an erster Stelle in str befindet
    char Name[ 255];
    char *pstr1, *pstr2;                                // Hilfs-Zeiger um str in die einzelnen Namen zerlegen zu können
    int nPersonCount;                           // Anzahl der Personen, die eingelesen werden

    bFirstIsKatrin = 0; bFirstIsRobert = 0; nPersonCount = 0; pFirstPerson = 0;
    pstr2 = str;

    do  // Diese Schleife wird für jeden zu extrahierenden Namen durchlaufen
    {
            nPersonCount++;                 // Anzahl der erkannten Personen um eins erhöhen
            memset( Name, 0, 255);          // Den String Namen mit nullen füllen
            pstr1 = strchr( pstr2, ',');    // pstr1 zeigt auf erstes Komma in pstr2
            if( pstr1 == 0) strcpy( Name, pstr2);       // falls es kein Komma gab, beinhaltet pstr2 nur einen Namen
            else strncpy( Name, pstr2, pstr1 - pstr2);  // ansonsten wird der erste Name extrahiert

            if( strcmp( Name, "Robert") == 0)           // Falls der gefunden Name "Robert" ist
            {
                    if( nGroup == 0)    // Gruppe 0 bedeutet, dass es sich um die Partygäste handelt und
                    {                   // es ist auszuschließen, dass Robert auf der Party war
                            printf( "Robert kann sich nicht unter den Partygästen befinden !!!\n");
                            return 0;  // Fehler
                    }
                    // Falls Robert der erste ist, wird dies durch bFirstIsRobert festgehalten
                    else if( nPersonCount == 1) bFirstIsRobert = 1;
                    // Falls Robert nicht der erste ist, muss die erste Person Robert kennen
                    else if( pFirstPerson) pFirstPerson->bKnowRobert = 1;
            }
            else if( strcmp( Name, "Katrin") == 0) // Falls der Name Katrin gefunden wird
            {
                    if( nPersonCount == 1) bFirstIsKatrin = 1; // und sich an erster Stelle befindet wir das vorgemerkt
            }
            else // Falls es sich weder um Robert noch um Katrin handelt
            {
                    // Falls dies der erste Name in str ist, zeigt pFirstPerson auf die erste Person
                    // dies ist wichtig, falls Robert auch noch in str enthalten ist und ihn die erste
                    // Person deshalb kennen muß (vgl. 7 Zeilen weiter oben)
                    if( nPersonCount == 1) pFirstPerson = AddPerson( Name, 0, nGroup);
                    // Wenn Robert der erste Name in str ist, muß jede Person Robert kennen und zu einer
                    // eigenen Gruppe gehören, da Robert nichts weiter erzählt
                    else if( bFirstIsRobert) AddPerson( Name, 1, nGroup++);
                    // Wenn Katrin der erste Name ist, muß jede Person zu einer eigenen Gruppe gehören
                    else if( bFirstIsKatrin) AddPerson( Name, 0, nGroup++);
                    else AddPerson( Name, 0, nGroup); // Ansonsten kennt die Person Robert nicht
            }
            // pstr2 zeigt nun auf den nächsten Namen in str
            // dadurch, dass 1 dazu addiert wird, wird das Komma übersprungen
            pstr2 = pstr1 + 1;
    }
    // sollte pstr1 == 0 sein, da kein Komma gefunden wurde und es sich somit um den letzten Namen
    // handelte, so wird die Schleife verlassen
    while( pstr1 != 0);
    return nPersonCount; // Die Anzahl der eingelesenen Person zurückgeben
}
```

```
int Input() // Liest alle Beziehungen und die Partygäste ein
{
    char str[ 255];
    int nPersons;           // nimmt den Rückgabewert von ReadGroup, also die Anzahl der gefundenen Persone, auf
    strcpy( str, "");       // str löschen
    nGroup = 0;             // mit Gruppe 0, also den Partygästen beginnen


    printf( "Geben Sie in der ersten Zeile die Partygäste nur durch Komma getrennt ein.\n");
    printf( "In den darauf folgenden Zeilen geben Sie bitte die Bekanntschaften an.\n");
    printf( "A,B,C,D bedeutet, daß die Person A die Personen B,C und D kennt\n");
    printf( "Zum Beenden ein kleines q eingeben\n");

    do
    {
            printf( "->"); // Eingabeaufforderung ausgeben
            if( gets( str) == 0) // falls beim Einlesen von str etw. schief geht
            {
                    printf( "Fehler bei Eingabe!\nDas Programm wird abgebrochen!\n");
                    return 1; // Fehler
            }
            // Falls ein q eingegeben wurde, wird das Einlesen beendet
            if( strcmp( str, "q") == 0) break;
            // Andernfalls wird str durch ReadGroup in die einzelnen Namen zerlegt
            // und diese in die Liste eingeordnet
            nPersons = ReadGroup( str);
            // Hier wird überprüft, ob die nötige Anzahl von Namen eingegeben wurde
            // bei nGroup == 0 handelt es sich um die Eingabe der Partygäste,
            // somit ist auch die Eingabe eines einzelnen Namens möglich, sonst müssen es mind. 2 sein.
            if( ((nPersons < 2) && (nGroup >= 1)) || ((nPersons < 1) && (nGroup == 0)))
            {
                    printf( "Fehler bei Eingabe!\nDas Programm wird abgebrochen!\n");
                    return 2; // Fehler
            }
            // GruppenIndex um eins erhöhen, damit die Personen bei der nächsten Eingabe
            // einer neuen Gruppe zugeordnet werden
            nGroup++;
    } while( 1);            // solange kein q eingegeben wird
    printf( "Das Einlesen wurde erfolgreich beendet.\n");
    return 0; // Fehlerfrei beendet
}
```

```
void Output()  // Gibt das Ergebnis aus
{
    int nPersonCount;     // Enthält die Anzahl der Personen, die Robert von der Party erzählen
    _Person *pPerson;     // Hilfszeiger zum Durchlaufen der Personenliste

    nPersonCount = 0;     // noch keine Personen gefunden, die Robert von der Party berichten
    printf( "\nRobert hört von folgenden Personen von der Party:\n");

    pPerson = pPersons;

    while( pPerson)         // solange es weitere Personen gibt
    {
            // Alle Personen werden ausgeben, die von der Party wissen und Robert kennen
            if((pPerson->nGroup == 0) && (pPerson->bKnowRobert == 1))
            {
                    printf( "%s  ", pPerson->Name); // Namen ausgeben
                    nPersonCount++;   // Person zählen
            }
            pPerson = pPerson->pNext; // zur nächsten Person gehen
    }
    // Ausgabe der Anzahl der Personen, von denen Robert von der Party erfährt
    printf( "\nInsgesamt: %d Person(en).\n", nPersonCount);
}

void Analyze() // Analysiert die Beziehungen
{
    _Person *pPerson1, *pPerson2, *pPerson3, *pPerson4;             // Hilfszeiger zum Durchlaufen der Persionenliste
    int bChanged, nFoundGroup;     // bChanged legt fest ob noch Liste zusammen gefasst wurden
    //pPerson1 = pPersons;
    bChanged = 1;
    while( bChanged)      // Wiederhole, solange noch Listen zusammen gefasst werden
    {
            bChanged = 0;          // bisher wurden noch keine Listen zusammen gefasst
            // Hier wird die gesamte Liste durchlaufen
            pPerson1 = pPersons;
            while( pPerson1)
            {
                    // Hier wird nochmals die gesamte Liste durchlaufen um Person1 mit Person2 vergleichen zu können
                    pPerson2 = pPersons;
                    while( pPerson2)
                    {
                            // Wenn eine Person2 mit dem gleichen Namen wie Person1 gefunden wird,
                            // so muß ihre Gruppe in die Gruppe von Person1 übernommen werden
                            // außer die Gruppe der Person2 ist 0
                            if(( strcmp( pPerson2->Name, pPerson1->Name) == 0) &&
                                                            (pPerson1 != pPerson2) && (pPerson2->nGroup != 0))
                            {
                                    bChanged = 1;
                                    // nFoundGroup speichert den Gruppenindex der Gruppe der Person2
                                    nFoundGroup = pPerson2->nGroup;
                                    // Falls Person2 Robert kannte, muß pPerson1 ihn auch kennen, da sie identisch sind
                                    if( pPerson2->bKnowRobert) pPerson1->bKnowRobert = 1;
                                    // Löschen der Person, da sie doppelt vorkommt
                                    if( pPerson2->pPrev) pPerson2->pPrev->pNext = pPerson2->pNext;
                                    if( pPerson2->pNext) pPerson2->pNext->pPrev = pPerson2->pPrev;
                                    pPerson4 = pPerson2->pNext;    // pPerson4 dient nur zum Zwischenspeichern
                                    delete pPerson2;                       // Person2 wird endgültig gelöscht
```

```
                                        pPerson2 = pPerson4;            // nun zeigt pPerson2 auf die nächste Person

                                        pPerson3 = pPersons;
                                        while( pPerson3) // In dieser Schleife werden alle Personen der Gruppe der Person2
                                        {           // in die Gruppe der Person1 übernommen
                                                if( pPerson3->nGroup == nFoundGroup)
                                                        pPerson3->nGroup = pPerson1->nGroup;
                                                pPerson3 = pPerson3->pNext; // weiter zur nächsten Person
                                        }
                                }
                                else pPerson2 = pPerson2->pNext; // weiter zur nächsten Person
                        }
                        pPerson1 = pPerson1->pNext; // weiter zur nächsten Person
                }
        }
}

void FreePersons() // Speicher von Personenliste freigeben
{
        _Person *pPerson; // Hilfszeiger zum Durchlaufen der Personenliste
        while( pPersons)
        {
                pPerson = pPersons->pNext;
                delete pPersons;                // Person löschen
                pPersons = pPerson;
        }
}

int main( void)
{
        if( Input()) return 1;  // Eingabe bearbeiten
        Analyze();              // Beziehungen analysieren
        Output();               // Ergebnis ausgeben
        FreePersons();          // Speicher freigeben
        return 0;
}
```

Ich erkläre hiermit, daß ich die Facharbeit ohne fremde Hilfe angefertigt habe.

Zorneding, den 15.1.1998

(Unterschrift des Schülers)

Bewertung:

Michael Wack

Bundeswettbewerb Informatik 1997 (1. Runde)

Zunächst ist die sehr ansprechende äußere Form dieser umfangreichen Arbeit hervorzuheben. Die ausgefeilten graphischen Darstellungen und der angenehme Blocksatz lassen keine Wünsche offen.

Bei den Lösungsideen und der Programmdokumentation stellt der Verfasser seine Strategie sehr verständlich und sinnvoll gegliedert dar. Sie berücksichtigen Sonderfälle und geben manchmal auch noch Ausblicke auf Verallgemeinerungen und präsentieren ihre Lösungen in makelloser Form. Der Verfasser versteht es in hervorragender Weise als kreativer Programmierer, seine Ideen umzusetzen und verständlich zu präsentieren.

Alle Programme wirken sehr gut strukturiert und auch optisch gut gegliedert. Jeder wesentliche Befehl ist kommentiert . Ein- und Ausgabe sind so verständlich, dass die Programme auch von nicht Eingeweihten bedient werden könnten.

Es handelt sich insgesamt um eine hervorragende Arbeit, die mit **15 Punkten = Note 1+** bewertet werden kann.

Ch. Vollmach