

30. Bundeswettbewerb Informatik

1. Runde

Raphael Michel

10. September 2011

Inhaltsverzeichnis

1	Junioraufgabe 2: Glücksrad	3
1.1	Zusammenfassung der Aufgabenstellung	3
1.2	Lösungsidee	3
1.3	Programmdokumentation	3
1.3.1	Eingabeformat	3
1.4	Programmablaufprotokoll	4
1.5	Programmtext	4
2	Aufgabe 1: Wer spiel fair?	6
2.1	Zusammenfassung der Aufgabenstellung	6
2.2	Lösungsidee	6
2.3	Programmdokumentation	7
2.4	Beispieldurchlauf	7
2.5	Programmtext	7
3	Aufgabe 4: Zaras dritter Fehler	9
3.1	Zusammenfassung der Aufgabenstellung	9
3.2	Lösungsidee	9
3.3	Lösungen	9
3.4	Programmdokumentation	10
3.5	Implementierung	11
3.6	Programmablaufprotokoll	11
3.6.1	Kompilieren	11
3.6.2	Vorbereiten	11
3.6.3	Code knacken	11
3.7	Programmtext	12
3.7.1	decrypt.cpp	12
3.7.2	Makefile	14
3.7.3	decodeall.sh	14

1 Junioraufgabe 2: Glücksrad

1.1 Zusammenfassung der Aufgabenstellung

Der Hauptgewinn bei einem Gewinnspiel mit einem drehbaren Rad wird dann ausgezahlt, wenn der Spieler das Rad so oft dreht wie es Felder besitzt und jedes Feld genau einmal trifft, wobei die Wahrscheinlichkeiten, dass das Rad sich bei einer Drehung um x Felder weiterbewegt gegeben sind. Das Programm soll für Anzahlen von bis zu 10 Feldern die Wahrscheinlichkeit berechnen, dass der Hauptgewinn erreicht wird.

1.2 Lösungsidee

Zur Beschreibung des Algorithmus gehen wir der Einfachheit halber von sechs Feldern aus, obwohl natürlich eine beliebige Anzahl eingesetzt werden kann.

Die möglichen Kombinationen der sechs Felder, die zum Hauptgewinn führen, entsprechen den *Permutationen*¹ der Menge $\{A, B, C, D, E, F\}$. Für jede dieser $6! = 720$ Möglichkeiten muss nun die Wahrscheinlichkeit errechnet werden, welche anschließend alle addiert werden, um das Gesamtergebnis zu halten.

1.3 Programmdokumentation

Um die mathematische Berechnung zu vereinfachen, werden die Felder des Rads im Algorithmus von $\{A, B, C, \dots\}$ zu $\{1, 2, 3, \dots\}$ umbenannt.

Für jede der Permutationen wird nun für jede Drehung berechnet, wie weit sich das Rad gedreht hat. Hierzu wird die passende Wahrscheinlichkeit aus der Tabelle ausgelesen und mit den Wahrscheinlichkeiten der anderen Schritte multipliziert. Die resultierende Wahrscheinlichkeit dieser einen Möglichkeit zum Hauptgewinn ergibt addiert mit denen aller anderen Permutationen das gesuchte Ergebnis, nämlich die Gesamtwahrscheinlichkeit, den Hauptgewinn zu erhalten.

Der Algorithmus ist in Python implementiert (lauffähig ab 2.3.x und unter 3.x).

1.3.1 Eingabeformat

Als Eingabeformat erwartet das Programm eine Textdatei, die für die in der Aufgabenstellung gegebenen Beispielwahrscheinlichkeiten folgendermaßen aussehen würde:

```
1;5/15
2;4/15
3;3/15
4;2/15
5;1/15
6;0/15
```

¹<http://de.wikipedia.org/wiki/Permutation>

Zur besseren Übersichtlichkeit wurde für die Wahrscheinlichkeiten eine Bruchdarstellung statt der Dezimaldarstellung gewählt (das Programm akzeptiert auch nur Brüche).

1.4 Programmablaufprotokoll

Die Bedienung des Programms erfolgt von der Kommandozeile und ist simpel: der einzige Parameter ist der Dateiname der Textdatei mit den Wahrscheinlichkeiten. Die einzige Ausgabe ist das Ergebnis, also die Wahrscheinlichkeit, den Hauptgewinn zu erreichen.

Beispiel mit den Daten aus der Aufgabenstellung:

```
$ ./berechnung.py eingabe.txt
0.0471703703704
```

Für ein Glücksrad mit sechs Feldern benötigt das Programm weit weniger als eine halbe Sekunde zur Berechnung. Für Glücksräder mit zehn Feldern steigt die Dauer stark an (auf dem Testgerät auf ca. 20 Sekunden), da nun statt $6! = 720$ Permutationen ganze $10! = 3628800$ Permutationen berechnet werden müssen – das ist in etwa das fünftausendfache.

1.5 Programmtext

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import sys
import itertools

# Wahrscheinlichkeitstabelle einlesen
if len(sys.argv) > 1:
    fname = sys.argv[1]
else:
    fname = 'eingabe.txt'
f = open(fname)
c = f.read().strip()
f.close()
lines = c.split("\n")
prob = {}
for line in lines:
    parts = line.split(";")
    fraction = parts[1].split("/")
    prob[int(parts[0])] = float(fraction[0])/float(fraction[1])
cnt = len(prob)
fields = range(1,cnt+1)

total = 0
# Permutationen
for p in itertools.permutations(fields):
    probability = 1
    last = 1
    for f in p:
        # Anzahl der Felder um die sich bewegt wird
```

```
    if f > last:
        distance = f - last
    elif f == last:
        distance = cnt
    else:
        distance = f + cnt - last
    # Zugehörige Wahrscheinlichkeit
    probability *= prob[distance]
    last = f
total += probability
print(total) # Endergebnis
```

2 Aufgabe 1: Wer spielt fair?

2.1 Zusammenfassung der Aufgabenstellung

Es wird bezweifelt, dass ein „Schere, Stein, Papier“ spielender Computer fair ist und seine Entscheidung unabhängig von der Eingabe des menschlichen Spielers getroffen wird. Es soll ein Verfahren entworfen werden, das es dem Computer ermöglicht, seine Fairness zu beweisen.

2.2 Lösungsidee

Um die Anforderungen zu erfüllen, muss der Computer die Wahl treffen *bevor* er die Eingabe des menschlichen Spielers erfährt. Um dies zu beweisen muss der Computer die Wahl folglich vor der Eingabe an einen Ort speichern, an dem er sie nicht mehr verändern kann, sie vom Spieler aber erst nach dem Spiel gelesen werden kann.

Ein einfacher Ansatz wäre, dass der Computer seine Entscheidung ausdrückt, der menschliche Spieler dann seine Wahl eingibt und erst dann den Ausdruck zu Gesicht bekommt (es wäre eine weitere Person oder Mechanik erforderlich, die sicherstellt, dass der Spieler den Ausdruck nicht zu früh sieht). Da ein Computer einen Ausdruck nicht mehr ändern kann, wäre dieses Verfahren sicher.

Da die Aufgabestellung eine praktikable und einfache Lösung verlangt, scheidet diese Möglichkeit aber erst einmal aus. Dem perfekten Verfahren muss also eine Bildschirmausgabe genügen, die der Spieler erst nach dem Spiel versteht.

Hierzu geben wir einen kryptologischen Hash aus. Für den Hash benutzen wir eine beliebige bekannte Hashfunktion, die in der Fachwelt allgemein als sicher anerkannt ist und die Kollisionen so gut wie möglich ausschließt. Beispielsweise kämen SHA-256, SHA-512¹ oder Whirlpool² in Frage.

Ausgegeben wird der Hash einer Zeichenkette, der sich aus der Wahl des Computers (Schere, Stein oder Papier) und einer zufällig generierten Zeichenkette (dem sog. *Salt*³) zusammensetzt. Dieser Salt wird vom Computer im Arbeitsspeicher gehalten, bis der menschliche Spieler ebenfalls seine Wahl getroffen hat. Danach gibt der Computer seine Wahl zusammen mit dem Salt auf dem Bildschirm aus. Der Spieler hat nun die Möglichkeit, selbst an einer Maschine seines Vertrauens oder mit einem Programm seines Vertrauens die Wahl des Computers und den Salt mit dem vereinbarten Algorithmus zu hashen und das Ergebnis mit dem vor dem Spiel vom Computer ausgegebenen Hash zu vergleichen.

Stimmen die Hashes überein, hat der Computer nicht betrogen. Unterscheiden sie sich, hat der Computer betrogen (oder es liegt in Fehler in der Implementierung der Hashfunktion vor).

Dieses Verfahren ist genauso sicher wie der Algorithmus und die Implementierung der Hashfunktion, die durch ihre Veröffentlichung und die Untersuchung durch Kryptologen der ganzen Welt als sehr sicher eingeschätzt werden können, wobei selbstverständlich keine Hashfunktion

¹<http://de.wikipedia.org/wiki/Secure.Hash.Algorithm>

²[http://de.wikipedia.org/wiki/Whirlpool_\(Algorithmus\)](http://de.wikipedia.org/wiki/Whirlpool_(Algorithmus))

³[http://de.wikipedia.org/wiki/Salt_\(Kryptologie\)](http://de.wikipedia.org/wiki/Salt_(Kryptologie))

der Welt davor sicher ist, Kollisionen zu enthalten. Eine gute Hashfunktion dürfte es jedoch unbewältigbar machen, in annehmbarer Zeit genau jene Kollision zu finden, die benötigt wird.

2.3 Programmdokumentation

Nachfolgend eine minimale Implementierung einer einzelnen Spielrunde in Python 2.

Das Programm gibt einen Hash aus (verwendeter Algorithmus: SHA-512), fragt nach der Wahl des Spielers und gibt danach seine Wahl, den Salt und das Spielergebnis aus.

Als Salt wird eine 64 Zeichen lange zufällige Kombination aus Buchstaben, Zahlen und Satzzeichen verwendet.

2.4 Beispieldurchlauf

```
Hash meiner Wahl: 95947b51ed472f24a11636995d97b8509c24538d0b48c15d847a7af82
21058c058a441e400831d62620b2390c4485a6d9c4a87fa3423af29eb1284aa3bd2a21a
Bitte triff deine Wahl (schere, stein oder papier): papier
Wahl des Computers: Stein
Gehashte Zeichenkette: Stein0hd08tbUFM*2l:U?2<h5CyE+l"lRR]c{qR7v9h.Z,Dy?-);
j'F~W3@Uxff~tRANR
Spieler gewinnt
```

2.5 Programmtext

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import hashlib
import random
import string

choices = ["Schere", "Stein", "Papier"]
choice = random.choice(choices) # Treffen der Wahl

salt = ''
chars = string.letters + string.digits + string.punctuation
# Beliebige anzeigbare Zeichen
for i in range(64): # 64 = Länge des Salts
    salt += random.choice(chars)

hashobj = hashlib.sha512()
hashobj.update(choice)
hashobj.update(salt)
hashstr = hashobj.hexdigest() # Hash als Hex-String
print "Hash meiner Wahl: %s" % hashstr # Ausgabe des Hashs

# Eingabe des Spielers
print "Bitte triff deine Wahl (schere, stein oder papier):",
```

```
inputstr = raw_input()
# folgendes Verfahren entscheidet, ob die Eingabe Schere, Stein oder Papier
# war, ohne dass die Groß-/Kleinschreibung des ersten Buchstabens eine
# Rolle spielt, denn wir schauen nur auf den zweiten :-))
if inputstr[1] == 'c': playerchoice = 'Schere'
elif inputstr[1] == 't': playerchoice = 'Stein'
elif inputstr[0] == 'p': playerchoice = 'Papier'
else: playerchoice = False

print "Wahl des Computers: %s" % choice
print "Gehashte Zeichenkette: %s" % "".join((choice, salt))

if playerchoice == choice:
    print "Unentschieden"
elif playerchoice == "Schere" and choice == "Stein":
    print "Computer gewinnt"
elif playerchoice == "Schere" and choice == "Papier":
    print "Spieler gewinnt"
elif playerchoice == "Stein" and choice == "Schere":
    print "Spieler gewinnt"
elif playerchoice == "Stein" and choice == "Papier":
    print "Computer gewinnt"
elif playerchoice == "Papier" and choice == "Stein":
    print "Spieler gewinnt"
elif playerchoice == "Papier" and choice == "Schere":
    print "Computer gewinnt"
```

3 Aufgabe 4: Zaras dritter Fehler

3.1 Zusammenfassung der Aufgabenstellung

Gegeben ist ein Buch sowie eine Zahlenfolge. Die Zahlenfolge beschreibt einen Klartext, indem ab dem Beginn einer Zeile im Buch jede Zahl der Zahlenfolge immer dem Abstand zum nächsten Buchstaben des Klartextes entspricht. Es sind jedoch weder die Zeile im Buch noch der Klartext bekannt. Das Programm soll in der Lage sein, 12 gegebene Zahlenfolgen mithilfe des Buches zu entschlüsseln. Leer- und Satzzeichen werden ignoriert¹.

3.2 Lösungsidee

Beim Verschlüsseln wird folgendermaßen vorgegangen: Wenn der Satz „Ohne Liebe keine Wahrheit“ verschlüsselt werden soll, nimmt man als erste Zahl die Position des ersten „O“ in der Buchzeile. Als zweite Zahl nimmt man den Abstand vom „O“ zum ersten „h“ nach dem ersten „O“. Daraus folgt: im Bereich zwischen dem Buchstaben, der von der ersten Zahl beschrieben wird und dem Buchstaben, der von der zweiten Zahl beschrieben wird, darf kein weiteres „h“ vorkommen, weil die zweite Zahl sonst nicht das *nächste* „h“ nach dem „O“ beschreiben würde.

An dieser Stelle greift mein Lösungsalgorithmus: Wir haben hierdurch jetzt eine Bedingung für Zeilen vorliegen, die den Klartext enthalten könnten und brauchen nur noch jede Zeile des Buches als Ausgangspunkt zu nehmen und zu testen, ob die Bedingung auf sie zutrifft. Wie sich zeigte, waren keine weiteren Maßnahmen nötig um die Treffer weiter einzuschränken, da mit den 12 Zahlenfolgen aus der Aufgabestellung jeweils genau eine Zeile gefunden wurde, die benutzt werden kann – und in dort fand sich dann auch der Klartext.

3.3 Lösungen

Die Zeilennummern beziehen sich auf die komplette Textdatei von bundeswettbewerb-informatik.de und nicht auf den reinen Buchtext.

zahlenfolge0.txt „Ohne Liebe keine Wahrheit“ ab Zeile 1321 (’>Ganz einfach. So ...’)

zahlenfolge1.txt „Das Denken soll man den Pferden überlassen, die haben den größeren Kopf“ ab Zeile 391 (’dem Bellingschen in ...’)

zahlenfolge2.txt „Wenn du irgendetwas verloren hast, nimm an du hättest es einem Armen gegeben“ ab Zeile 467 (’auf das jugendlich ...’)

zahlenfolge3.txt „Der Ellenbogen ist dem Munde nahe dennoch kann man nicht selbst hineinbeißen“ ab Zeile 47 (’und Gartenseite hin ...’)

¹Anm.: Es steht zwar nicht in der Aufgabe, jedoch zeigen die Ergebnisse, dass Zahlen ebenfalls ignoriert werden müssen

zahlenfolge4.txt „Wovon das Herz voll ist, davon geht der Mund über“ ab Zeile 347 (‘Boot und ließen von...’)²

zahlenfolge5.txt „Je verdorbener der Staat, desto mehr Gesetze hat er“ ab Zeile 67 (‘angenehmen und zuge...’)

zahlenfolge6.txt „Erinnere dich, Mensch, dass du aus Staub bist und zu Staub zurückkehren wirst“ ab Zeile 580 (‘»Gott, Effi, wie du...’)

zahlenfolge7.txt „Wer den Aal hält bei dem Schwanz, dem bleibt er weder halb noch ganz“ ab Zeile 768 (‘Wollen wir...’)

zahlenfolge8.txt „Schlechte Handwerker klagen über ihr Werkzeug“ ab Zeile 220 (‘»Aber du sagtest...’)

zahlenfolge9.txt „Wenn man durch Zweifel läuft ist der Weg zum Himmel lang“ ab Zeile 81 (‘Auge von der Arbeit ...’)

zahlenfolgeA.txt „Jede Dummheit leidet am Ekel vor sich selbst“ ab Zeile 159 (‘wäre doch gegangen...’)

zahlenfolgeB.txt „Ein guter Spruch ist die Wahrheit eines ganzen Buches in einem einzigen Satz“ ab Zeile 381 (‘haben. Übrigens, ...’)

3.4 Programmdokumentation

Zur Vorbereitung wird das Buch mithilfe des Unix-Kommandos *sed* von allen Leerzeichen, Satzzeichen und Zahlen befreit (dies wird per GNU *make*-Befehl automatisch erledigt).

Das ausführbare Programm, *decrypt* (implementiert in C++), entschlüsselt letztendlich die Zahlenfolgen. Dazu lädt es zuerst das bereits umgewandelte Buch (dass mittlerweile noch von einem Skript nach ISO-8859-1 umgewandelt wurde, um Probleme mit deutschen Umlauten zu vermeiden) unterteilt nach Zeilen in den Arbeitsspeicher (in Form eines **vector**).

Danach berechnet es die Summe aller Zahlen in der codierten Zahlenfolge, denn dies entspricht der Länge des Buchabschnittes, in dem die Lösung versteckt ist.

Anschließend geht es in einer Schleife Zeile für Zeile des Buches durch. Zuerst lädt es so viele folgende Zeilen zur Berechnung hinzu, bis die erforderte Länge, die im vorherigen Schritt als Summe errechnet wurde, erreicht wird.

Danach wird von diesem Zeilenanfang ausgehend jede Zahl der Zahlenfolge verarbeitet und untersucht, ob der resultierende Buchstabe im übersprungenen Teil der Zeile bereits vorkam. Ist dies der Fall, wird abgebrochen und mit der nächsten Zeile weitergearbeitet.

Tritt dieser Fall nicht ein, handelt es sich um eine formal gültige Lösung, die dem Benutzer am Ende präsentiert wird, damit er mit menschlichem Verstand entscheiden kann, ob es sich um einen deutschen Satz handelt. Die Ergebnisse haben allerdings gezeigt, dass das Verfahren die gesuchten Sätze sehr zuverlässig und zweifelsfrei findet und der Mensch außer bei *zahlenfolge4.txt* (siehe dortige Fußnote) nichts zu entscheiden hat.

²Anm.: Es gab bei dieser Zahlenfolge tatsächlich eine weitere Zeile, auf die die Bedingungen des Algorithmus zutrafen, aber da die Lösung „kleBrbkcounsootodtstoeonteoranedboomh“ lautete und in den englischsprachigen Copyrightinweisen des Verlags gefunden wurde, habe ich sie ignoriert.

3.5 Implementierung

Das Verfahren ist in C++, Shellskripten und GNU make implementiert. Die Implementierung ist auf ein Linux-System ausgelegt. Theoretisch müsste ein Kompilieren mit MinGW und GNU-Win32 auch auf Windows möglich sein, dies habe ich aber nicht getestet.

3.6 Programmablaufprotokoll

3.6.1 Kompilieren

Das Programm benötigt um kompiliert und benutzt zu werden:

- Einen C++-Compiler, z.B. `g++`³
- Die C++-Bibliothek `libboost`⁴
- GNU `make`⁵
- `iconv`⁶ als Kommandozeilenbefehl⁷
- GNU `sed` stream editor (`sed`)⁸
- Eine Kommandozeile mit Unix-Shell (auf jedem Unix-System vorhanden)

Das Kompilieren geht mit einem Befehl:

```
$ make
```

3.6.2 Vorbereiten

Mit dem folgenden Befehl wird das Buch (das als `book.txt` im selben Ordner liegen sollte) in das benötigte Format gebracht. Dazu werden zuerst alle Sonderzeichen aus dem Buch entfernt und danach das Buch von UTF-8 nach ISO-8859-1 umgewandelt. Das Ergebnis liegt dann als `book.dat` im selben Ordner.

```
$ make book
```

3.6.3 Code knacken

Der folgende Befehl erwartet als Parameter das Originalbuch (`book.txt`), das umgewandelte Buch (`book.dat`) und den Zahlencode, der entschlüsselt werden soll. Die einzelnen Zahlen können durch Leerzeichen, Kommata, Punkte oder Bindestriche getrennt werden.

```
$ ./decrypt book.txt book.dat \  
    "13,34,7,13,11,3,9,58,1,93,4,1,1,1,4,21,7,3,23,5,5,15"  
ohneliebekeinewahrheit (ab Zeile 1321: '»Ganz einfach. So g...')
```

³<http://gcc.gnu.org>, Ubuntu-Paket: `g++`

⁴<http://www.boost.org>, Ubuntu-Paket: `libboost1.42-dev`

⁵<http://www.gnu.org/software/make>, Ubuntu-Paket: `make`

⁶<http://www.gnu.org/software/libiconv>

⁷enthalten im Ubuntu-Paket `libc-bin`

⁸<http://sed.sourceforge.net>, vorhanden auf jedem Unix-System

Die Ausgabe beinhaltet die Lösung und die Fundstelle. Gibt es mehrere mögliche Lösungen, werden alle ausgegeben.

Es liegt ein Shellskript bei, welches das Programm automatisch mit den 12 Zahlenfolgen füttert, die gelöst werden müssen. Dazu müssen diese als .txt-Dateien im Unterordner `zahlenfolgen/` liegen.

```
$ ./decodeall.sh book.txt book.dat zahlenfolgen
zahlenfolgen/zahlenfolge0.txt
ohneliebekeinewahrheit (ab Zeile 1321: '»Ganz einfach. So g...')
zahlenfolgen/zahlenfolge1.txt
dasdenKensollmandenpferdenüberlassendiehabendengrößerenkopf (ab
Zeile 391: 'dem Bellingschen in ...')
```

3.7 Programmtext

3.7.1 decrypt.cpp

Das Programm `decrypt` leistet die eigentliche Arbeit und findet die Lösungen.

```
#include <iostream>
#include <string>
#include <fstream>
#include <map>
#include <set>
#include <vector>
#include <boost/algorithm/string.hpp>
using namespace std;
using namespace boost;

int main(int argc, char** argv)
{
    if(argc != 4){
        cout << "entschlüsselt den Code" << endl;
        cout << "    ./decrypt book compiledbook cypher" << endl;
        return 0;
    }
    // Cyphertext von der Kommandozeile lesen auf aufspalten
    vector<string> cypherin;
    split(cypherin, argv[3], is_any_of(" ,.-"));
    int cypherlen = cypherin.size();
    // Cyphertext in ein Array aus Ganzzahlen umwandeln
    int cypher[cypherlen];
    for (int h = 0; h < cypherin.size(); h++){
        cypher[h] = atoi(cypherin[h].c_str());
    }

    // Buch einlesen, um zu den Ergebnissen die Zeilenanfänge
    // ausgeben zu können.
    ifstream infile1 (argv[1], ifstream::binary);
    vector<string> obook;
```

```
string temp;
while( getline( infile1, temp ) ) {
    obook.push_back(temp);
}

// kompiliertes Buch einlesen
ifstream infile2 (argv[2], ifstream::binary);
vector<string> book;
while( getline( infile2, temp ) ) {
    book.push_back(temp);
}

// Länge des Textes in dem die Lösung versteckt ist
// (= Summe der Codezahlen)
int excerptlen = 0;
for (int i = 0; i < sizeof(cypher)/sizeof(int); i++){
    excerptlen += cypher[i];
}

// Buch durchsuchen
int a = 0;
int last = 0;
int i = 0;
string chr;
string plain;
set<string> solutions;
map<string,int> positions;

bool failed = false;
for (int l = 0; l < book.size(); l++){ // Zeile für Zeile versuchen
    failed = false;
    temp.clear();
    plain.clear();
    a = 0;
    // Normalerweise erstreckt sich der Code über mehrere Zeilen,
    // deshalb müssen wir die folgenden Zeilen mit beachten.
    while(temp.length() <= excerptlen and l+a < book.size()){
        if(l+a < book.size())
            temp.append(book[l+a]);
        a++;
    }
    last = 0;
    i = 0;
    // Zeichen im richtigen Abstand suchen
    for (int j = 0; j < cypherlen; j++){
        i += cypher[j];
        chr = temp[i-1];
        if(temp.find(chr, last) < i-1){
            // Es hätte eine bessere Lösung für Zara gegeben
```

```

        // -> keine Fundstelle
        failed = true;
        break;
    }else{
        plain += chr;
        last = i;
    }
}
if(!failed and plain.length() == cypherlen){
    // Es handelt sich um eine Fundstelle
    solutions.insert(plain);
    positions[plain] = 1;
}
}

// Ausgabe der Ergebnisse
for (set<string>::iterator it=solutions.begin();it!=solutions.end();it++)
{
    temp = *it;
    cout << temp;
    int l = positions[temp];
    cout << " (ab Zeile " << l+1 << ": ";
    cout << obook[l].substr(0,20) << "...')" << endl; // Zeilenanfang
}
return 0;
}

```

3.7.2 Makefile

Die Anweisungen für GNU make

```

build: decrypt
decrypt.o:
    g++ -c decrypt.cpp
decrypt: decrypt.o
    g++ decrypt.o -o decrypt
clean:
    rm *.o
book:
    sed 's/[ .,:;><!?()*%$"0-9]//g' book.txt | sed "s/-//g" | \
    sed "s/'//g" | sed "s/[//g" | sed "s/\\//g" > book.tmp
    iconv -f UTF-8 -t ISO_8859-1 book.tmp > book.dat
    rm book.tmp

```

3.7.3 decodeall.sh

```

#!/bin/sh
# ./decodeall.sh book.txt book.dat zahlenfolgen
# Dekodiert alle Zahlenfolgen im Ordner zahlenfolgen/
# Ausgabe: Dateiname<LF>Lösung

```

```
for f in $3/*.txt; \  
do \  
    echo $f; \  
    ./decrypt $1 $2 "$(cat $f)"; \  
done
```