Lösungsideen der Aufgaben des 26. Bundeswettbewerb Informatik

2. Runde

Markus Schröder

6. April 2008

Inhaltsverzeichnis

1. Aufgabe 1: Berufsinformationstag

1.1. Lösungsidee 3
1.2. Programm-Dokumentation 4
1.3. Programm-Text 5
1.4. Aufgaben 12
1.4.1. Aufgabe 1 12
1.4.2. Aufgabe 2 12

2. Aufgabe 3: Geheimnisvolle Töne

2.1. Lösungsidee 13-16
2.2. Programm-Dokumentation 17
2.3. Programm-Ablaufprotokolle 18
2.4. Programm-Text 19-26

1. Aufgabe 1: Berufsinformationstag

1.1. Lösungsidee

Das Problem muss in Mathematischer Form gefasst werden:
Für einen Schüler nimmt man die Variable S_i . *i* steht für den Index des Schülers. Da die Teilnehmer von der Anzahl maximal 240 und minimal 80 sein dürfen, gilt $i \in [80,240]$. Jeder Schüler hat eine Prioritätsliste P_j , wobei $j \in [0,5]$, da es 6 Prioritäten gibt. Dann gibt es noch die Variable B_k . *k* steht für jeden einzelnen Beruf, den der Schüler wählen kann. Also gilt bei 16 Berufen $k \in [0,15]$. Wenn ein Schüler ein Beruf gewählt hat, dann ist dort der Wert 1 eingetragen, andernfalls 0: Es gilt $B_k \in \{0,1\}$.

Ein Beispiel:
Wenn der 3. Schüler in seinem 1. Eintrag den 12 . Beruf einträgt, dann lautet die Formel $S_2 P_0 B_{11} = 1$.

Für jeden Beruf soll es eine maximale und minimale Teilnehmeranzahl geben. Dafür gibt es die Variablen $maxT_k$ und $minT_k$. *k* steht wieder für jeden Beruf.

Jeder Schüler darf in seiner List für eine Priorität nur einen Beruf wählen:

$$S_i P_j \sum_{k=0}^{15} B_k = 1$$

.

Also muss gelten: $S_i \sum_{j=0}^{5} P_j \sum_{k=0}^{15} B_k = 6$, damit der Schüler 6 Berufe gewählt hat.

Um die Teilnehmeranzahl festzulegen gibt es die Variable *t*.

Wenn man herausfinden will, wie viele Schüler den 6. Beruf in ihrem 1. Eintrag gewählt haben, dann lautet die Formel bei *t* Teilnehmer:

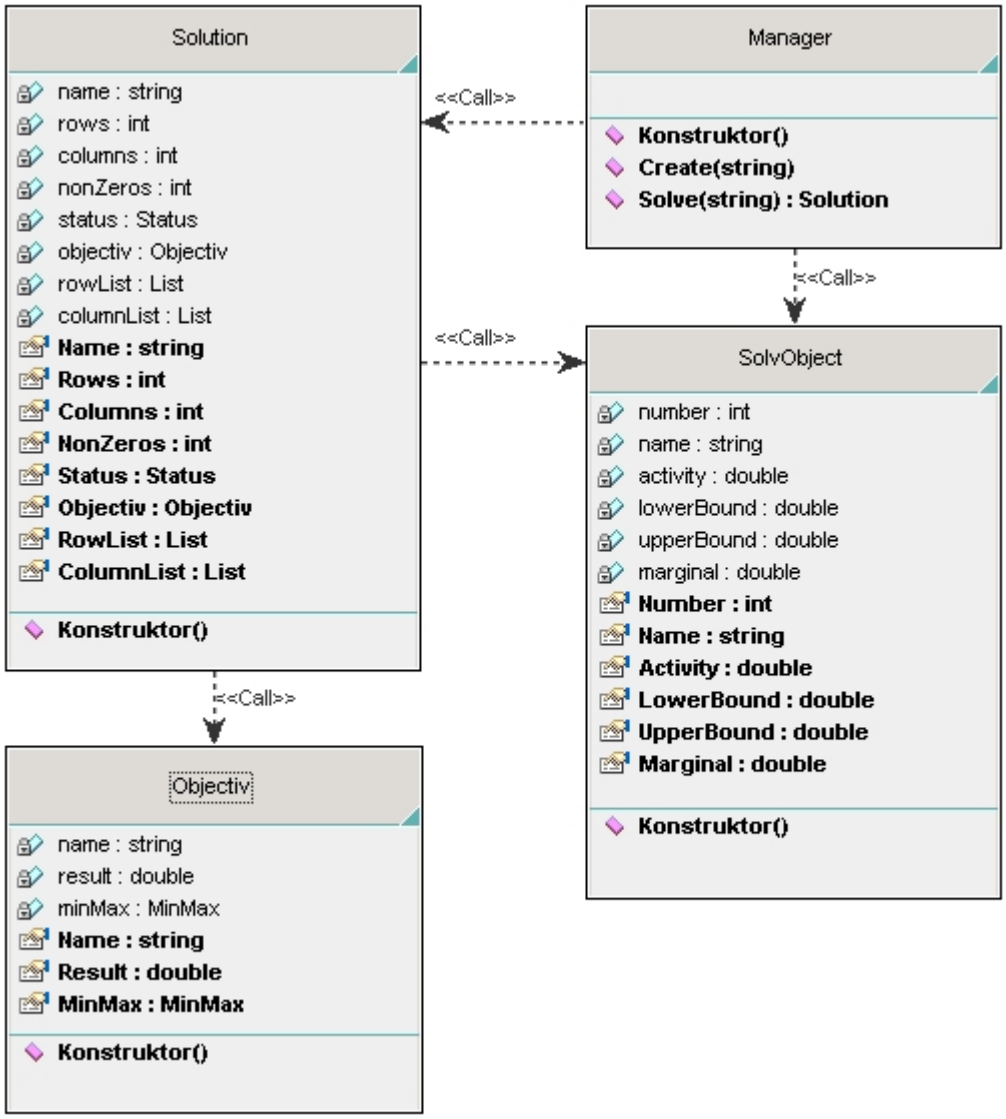
$$\sum_{i=0}^{t} S_i P_0 B_5 = a$$

.

Das Ziel ist es, über der minimal Grenze und unter der maximal Grenze zu liegen. Dann lautet also die Ausgangsformel:

$$maxT_k \geqslant \sum_{i=0}^{t} S_i P_j B_k \leqslant minT_k \quad , \quad k \in [0,15],\ j \in [0,5] \quad .$$

1.2. Programm-Dokumentation



Es gibt die Hauptklasse *Manager* und drei Datenklassen: *Solution, SolvObject* und *Objectiv.*

Die *Manager*-Klasse hat 2 Methoden:
Die *Create(string)*-Methode erstellt aus dem Wahlergebnis der Teilnehmer eine Mathprog-Datei.

Diese wird unter dem angegeben Namen gespeichert.

Die *Solve(string) : Solution*-Methode führt die Datei „glpsol.exe“ aus, die dann die erstellte Datei ausrechnet. Nachdem die „glpsol.exe“ fertig gerechnet hat, erstellt sie eine „output.txt“ in der das Ergebnis der Rechnung steht.
Nun wird das Ergebnis heraus gefiltert und in die Datenklasse *Solution* gespeichert und zurückgegeben.

1.3. Programm-Text

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using System.IO;
using System.Threading;

namespace Berufsinformationstag
{
   class Manager
   {
      public Manager()
      {

      }

      public void Create(string modelDataFilename)
      {
         StreamWriter w = new StreamWriter(modelDataFilename);

         w.WriteLine("var x, <= 5;");
         w.WriteLine();
         w.WriteLine("maximize obj: x;");
         w.WriteLine();
         w.WriteLine("end;");
         w.WriteLine();

         w.Close();
      }

      public Solution Solve(string modelDataFilename)
      {
         if (!File.Exists("glpsol.exe"))
            throw new Exception("Die Datei glpsol.exe existiert nicht!");

         if (!File.Exists(modelDataFilename))
            throw new Exception("Die Datei " + modelDataFilename + " existiert nicht!");

         //errechnen und in output speichern
         Process p = Process.Start("glpsol.exe", "-m " + modelDataFilename + " -o output.txt");
```

```
while (!p.HasExited)
{
   Thread.Sleep(1);
}

if (!File.Exists("output.txt"))
   throw new Exception("Fehler bei der Berechnung!");

//output lesen und in sol schrieben
Solution sol = new Solution();

string[] lines = File.ReadAllLines("output.txt");

for (int i = 0; i < lines.Length; i++)
{
   if (lines[i].Contains("Problem"))
   {
      sol.Name = lines[i].Split(new char[] { ':' })[1].Trim();
   }
   else if (lines[i].Contains("Rows"))
   {
      sol.Rows = int.Parse(lines[i].Split(new char[] { ':' })[1]);
   }
   else if (lines[i].Contains("Columns"))
   {
      sol.Columns = int.Parse(lines[i].Split(new char[] { ':' })[1]);
   }
   else if (lines[i].Contains("Non-zeros"))
   {
      sol.NonZeros = int.Parse(lines[i].Split(new char[] { ':' })[1]);
   }
   else if (lines[i].Contains("Status"))
   {
      string stat = lines[i].Split(new char[] { ':' })[1].Trim();

      switch (stat)
      {
         case "OPTIMAL":
            sol.Status = Status.Optimal;
            break;

         default:
            sol.Status = Status.Other;
            break;
      }
   }
   else if (lines[i].Contains("Objective"))
   {
      string[] obj = lines[i].Split(new char[] { ':' })[1].Split(new char[] { '=' });

      string[] obj2 = obj[1].Split(new char[] { '(', ')' });

      sol.Objectiv.Name = obj[0].Trim();
      sol.Objectiv.Result = double.Parse(obj2[0]);

      if (obj2[1].Trim() == "MAXimum")
         sol.Objectiv.MinMax = MinMax.MAXimum;
      else if (obj2[1].Trim() == "MINimum")
```

```
            sol.Objectiv.MinMax = MinMax.MINimum;
        }
        else if (lines[i].Contains("----"))
        {
            i += 1;

            //ROWS
            while (lines[i] != "")
            {
                SolvObject obj = new SolvObject();

                obj.Number = int.Parse(lines[i].Substring(0, 7));
                obj.Name = lines[i].Substring(7, 12).Trim();

                string act = lines[i].Substring(23, 13).Trim();
                if (act != "")
                    obj.Activity = double.Parse(act);

                string lb = lines[i].Substring(37, 13).Trim();
                if (lb != "")
                    obj.LowerBound = double.Parse(lb);

                string ub = lines[i].Substring(51, 13).Trim();
                if (ub != "")
                    obj.UpperBound = double.Parse(ub);

                string mar = lines[i].Substring(65).Trim();
                if (mar != "")
                    obj.Marginal = double.Parse(mar);

                sol.RowList.Add(obj);

                i++;
            }

            i += 3;

            //COLUMNS
            while (lines[i] != "")
            {
                SolvObject obj = new SolvObject();

                obj.Number = int.Parse(lines[i].Substring(0, 7));
                obj.Name = lines[i].Substring(7, 12).Trim();

                string act = lines[i].Substring(23, 13).Trim();
                if (act != "")
                    obj.Activity = double.Parse(act);

                string lb = lines[i].Substring(37, 13).Trim();
                if (lb != "")
                    obj.LowerBound = double.Parse(lb);

                string ub = lines[i].Substring(51, 13).Trim();
                if (ub != "")
                    obj.UpperBound = double.Parse(ub);

                string mar = lines[i].Substring(65).Trim();
```

```
                if (mar != "")
                    obj.Marginal = double.Parse(mar);

                sol.ColumnList.Add(obj);

                i++;
            }
        }
    }

    return sol;
  }
 }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Berufsinformationstag
{
  public enum Status
  {
    Optimal, Other
  }
  public enum MinMax
  {
    MAXimum, MINimum
  }

  class Solution
  {
    private string name;
    private int rows;
    private int columns;
    private int nonZeros;
    private Status status;
    private Objectiv objectiv;

    private List<SolvObject> rowList;
    private List<SolvObject> columnList;

    public Solution()
    {
      objectiv = new Objectiv();
      rowList = new List<SolvObject>();
      columnList = new List<SolvObject>();
    }

    public string Name
    {
      get
```

```
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    public int Rows
    {
        get
        {
            return rows;
        }
        set
        {
            rows = value;
        }
    }
    public int Columns
    {
        get
        {
            return columns;
        }
        set
        {
            columns = value;
        }
    }
    public int NonZeros
    {
        get
        {
            return nonZeros;
        }
        set
        {
            nonZeros = value;
        }
    }
    public Status Status
    {
        get
        {
            return status;
        }
        set
        {
            status = value;
        }
    }
    public Objectiv Objectiv
    {
        get
        {
            return objectiv;
        }
```

```
    }
    public List<SolvObject> RowList
    {
      get
      {
        return rowList;
      }
    }
    public List<SolvObject> ColumnList
    {
      get
      {
        return columnList;
      }
    }
  }

  class Objectiv
  {
    private string name;
    private double result;
    private MinMax minMax;

    public Objectiv()
    {}

    public string Name
    {
      get
      {
        return name;
      }
      set
      {
        name = value;
      }
    }
    public double Result
    {
      get
      {
        return result;
      }
      set
      {
        result = value;
      }
    }
    public MinMax MinMax
    {
      set
      {
        minMax = value;
      }
      get
      {
        return minMax;
      }
```

```
    }
}

class SolvObject
{
    private int number;
    private string name;
    private double activity;
    private double lowerBound;
    private double upperBound;
    private double marginal;

    public SolvObject()
    {
        activity = double.NaN;
        lowerBound = double.NaN;
        upperBound = double.NaN;
        marginal = double.NaN;
    }

    public int Number
    {
        get
        {
            return number;
        }
        set
        {
            number = value;
        }
    }
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    public double Activity
    {
        get
        {
            return activity;
        }
        set
        {
            activity = value;
        }
    }
    public double LowerBound
    {
        get
        {
            return lowerBound;
```

```
        }
        set
        {
            lowerBound = value;
        }
    }
    public double UpperBound
    {
        get
        {
            return upperBound;
        }
        set
        {
            upperBound = value;
        }
    }
    public double Marginal
    {
        get
        {
            return marginal;
        }
        set
        {
            marginal = value;
        }
    }
}}
```

1.4. Aufgaben

1.4.1. Aufgabe 1

Jeder Veranstaltung hat eine minimale Teilnehmeranzahl und eine maximale Teilnehmeranzahl. Wenn zu viele Teilnehmer in eine Veranstaltung wollen, dann müssen Teilnehmer ab gehen, also ihre 2. Wahl hinnehmen, oder der Kurs wird aufgesplittet, damit er 2 Mal gehalten werden kann. Ein Kurs kann aber nur maximal 4 Mal gehalten werden. Wenn 20 Leute in eine Veranstaltung gehen wollen, die aber nur maximal 10 Leute zulässt, dann kann man die 20 Leute aufteilen und in 2 Veranstaltungen unterbringen.

Die Teilnehmer sollen mit folgenden Kriterien zugeordnet werden:

- Wenn ein Teilnehmer wechseln muss, also seine 1. Wahl gegen die 2. Wahl austauschen muss, dann wäre ein Teilnehmer zu wählen, der durch das Wechseln nicht in einen Kurs gerät, der schon überfüllt ist. Es sollte also jemand sein, der in einen Kurs gehen soll, der sehr wenige Teilnehmer hat.

- Die Veranstaltungen sollten nicht überfüllt, aber auch nicht an Teilnehmer mangeln. Also wären die Teilnehmer so einzuordnen, dass die Anzahl möglichst genau zwischen minimaler und maximaler Teilnehmeranzahl liegt.

- Wie schon erwähnt, soll so wenig wie möglich gewechselt werden. Das heißt, wenn ein

Teilnehmer wechseln muss, dann sollte das nur 1. gegen 2., 2. gegen 3., 3. gegen 4., 4. gegen 5., 5. gegen 6. Wahl sein, da das die geringsten Umstellungen für ein Teilnehmer sind.

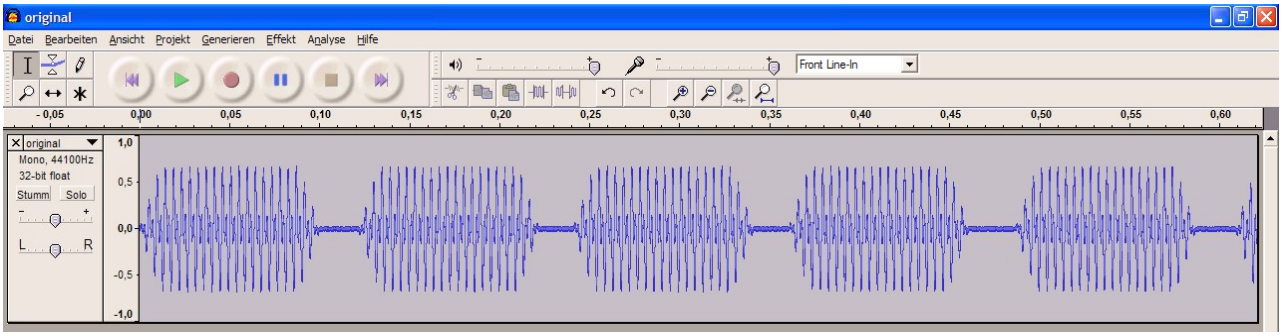
1.4.2. Aufgabe 2

Die Elternteile teilen vorher mit, wie viel sie maximal und minimal zu ihren Veranstaltungen zulassen. Wenn die Teilnehmer gewählt haben, muss gezählt werden, wie viele wo hin wollen. Das muss dann mit der maximalen und minimalen Teilnehmeranzahl der Veranstaltungen passen.

2. Aufgabe 3: Geheimnisvolle Töne

2.1. Lösungsidee

Analyse



Zuerst habe ich mir die Wave-Datei angehört. Dabei ist mir das immer stärker werdende Rauschen aufgefallen und ich war der Meinung, dass jeder Ton gleich ist.
Danach habe ich die Wave-Datei mit dem Programm *Audacity* geöffnet.

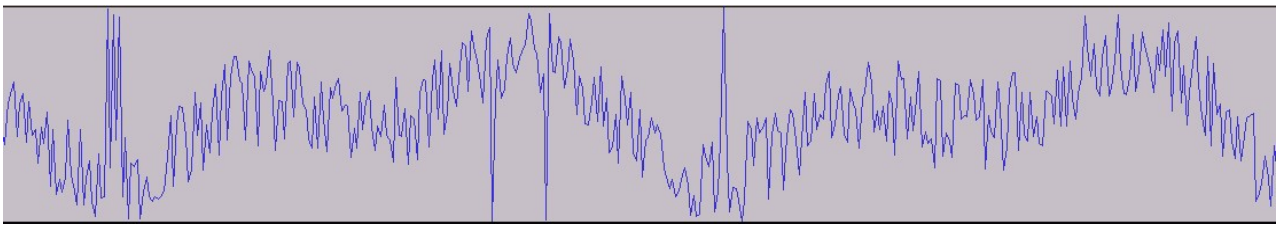
Ich schaute mir die Tonspur genau an, konnte mir aber noch kein Reim darauf machen, wie die Nachricht verschlüsselt ist. Erst als ich die 8 Töne aneinander gereiht sah, fiel es mir auf: Die 8 Töne repräsentieren ein Byte. Das müsste bedeuten, dass die 8 Töne unterschiedlich sein mussten. Bei näherem Betrachten fiel mir auf, dass die Töne wirklich unterschiedlich sind.

Ich habe festgestellt, dass die Töne unterschiedliche Schwingungen haben.
Die eine Schwingung hat in der Reihenfolge: ein großes Minimum, ein kleines Maximum, ein kleines Minimum und ein großes Maximum.
Bei der anderen Schwingung hingegen: ein großes Minimum, ein großes Maximum, ein kleines Minimum und ein kleines Maximum.
Diese unterschiedlichen Schwingungen repräsentieren Bits.
Die eine Schwingung das Bit 0 und die andere das Bit 1.

Mit diesen Entdeckungen probierte ich erst einmal, nur durch sehen, eine Bit-folge aufzuschreiben. Die Bits wandelte ich in Bytes um und dann in Chars. Das Wort das ich dabei herausfand, war „Dies“. Nun wusste ich, wie die Nachricht verschlüsselt war und dachte über ein Algorithmus nach, der diese Nachricht herausfinden kann.

Das Rauschen

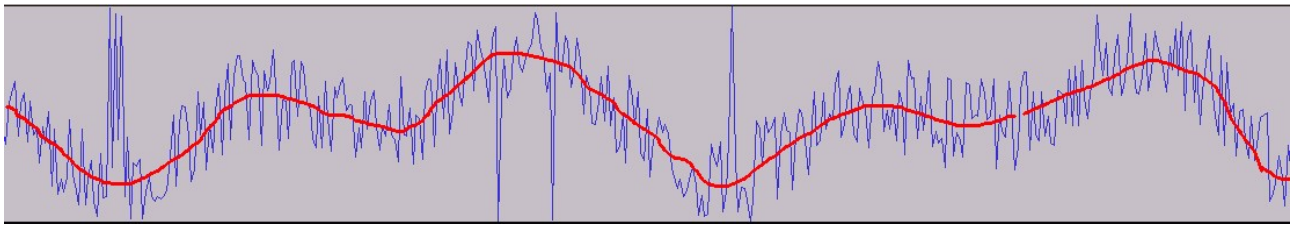
Mir war gleich bewusst, dass ich das Rauschen, entweder in den Algorithmus mit einberechnen oder das Rauschen vermindern muss. Ich entschied mich für die Rauschverminderung.



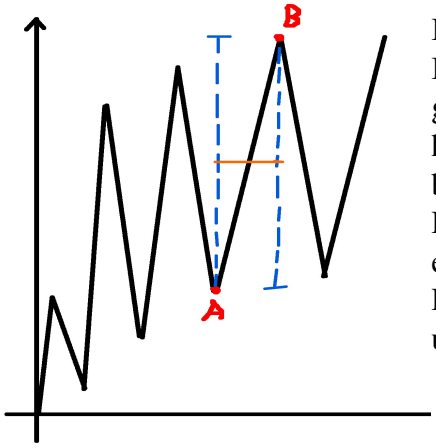
Das Rauschen der Tonspur

Wie man auf der Abbildung erkennen kann, ist das Signal verrauscht. Würde man eine Linie direkt zwischen den maximalen Ausschlägen und den minimalen Ausschlägen ziehen, würde man ungefähr das Originalsignal erhalten.

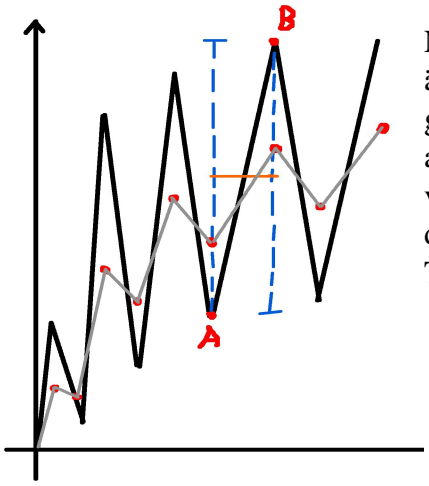
Die rote Linie ist die Mitte der Extremas



In der Abbildung kann man erkennen, das somit das Originalsignal fast hergestellt wird.
Um dies zu erreichen habe ich mir folgendes überlegt:

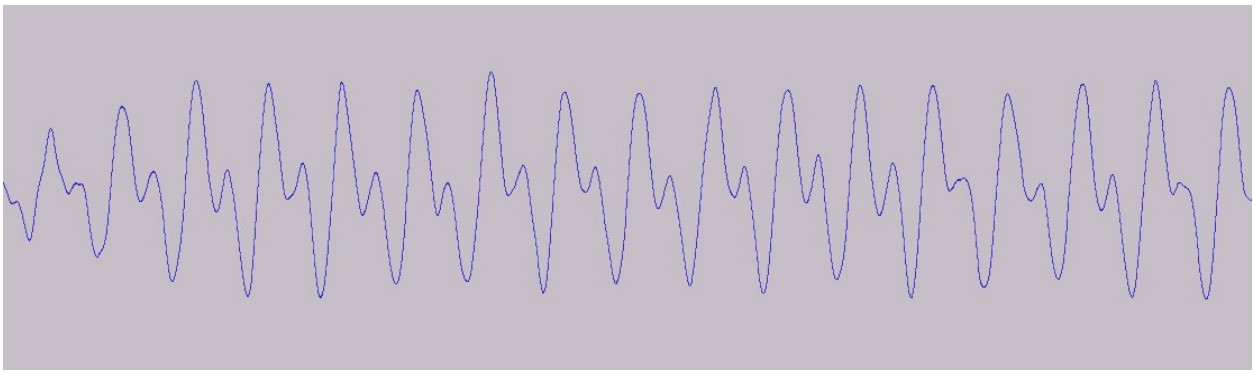


In der Darstellung wird das Rauschen symbolisch dargestellt. Die Idee ist, durch Annäherung der Punkte A und B das Signal zu glätten. Dabei wird der Abstand von A zu B bzw. B zu A halbiert. Die Punkte werden dann um diesen Wert nach oben (A) bzw. nach unten (B) verschoben.
Danach wird das selbe mit den nächsten Punkten gemacht. So entsteht ein etwas besseres Signal, bei dem aber immer noch ein Rauschen stark zu hören ist. Deshalb wende ich dieses Verfahren ungefähr 50 Mal an, bis das Signal geglättet ist.



Nach einem ersten Durchlauf müsste das Signal der Grauen Linie ähneln. Schon jetzt kann man erkennen, dass das Signal deutlich glätter geworden ist. Der Durchlauf beginnt am Anfang und endet am Ende der Tonspur. Es werden immer Paare von Punkten verglichen, wobei der rechte Punkt des Paares wieder ein Paar mit dem nächsten Punkt bildet. So zieht sich dies durch die ganze Tonspur.

Hier sieht man das Originalsignal, nachdem es 50 Mal mit der Methode verändert wurde:



Die Abbildung ist von dem Ende der Tonspur (dort, wo das Rauschen am stärksten ist)

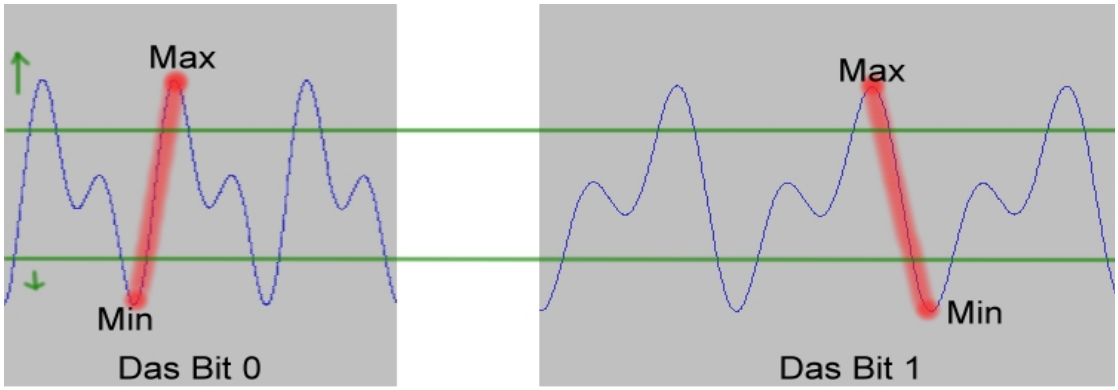
Man kann erkennen, dass das Originalsignal gut erkennbar ist.

Wave

Da die Nachricht in einer Wave-Datei steckt, ist es wichtig zu wissen, wie eine Wave-Datei aufgebaut ist.
Die Daten beginnen in der Wave-Datei bei dem Offset 44.
Auf dem Aufgabenblatt wird erklärt, dass bei dieser Wave-Datei ein Daten-Wort 16 Bit, also 2 Byte groß ist. Umgewandelt ergeben die 2 Byte ein Short. Die Shorts drücken aus, wie hoch bzw. wie tief das Signal verläuft. Ein Short kann maximal 32767 groß und minimal -32768 klein sein.

Die Nachricht

Es muss, wie bei der Analyse schon erklärt, die Reihenfolge der Extremas herausgefunden werden, damit man ein Ton als Bit, 8 Bits als Byte, das Byte als Char und die Chars als Nachricht interpretieren kann.



Also werden die Extremas, Maximal und Minimal, gesucht. Hierbei geht das Programm alle Werte durch und schaut, wo die Steigung wieder in die andere Richtung geht. Wenn dieser Wert über der Maximumgrenze (im Bild grün eingezeichnet) ist, kann dieser Wert als Maximum bezeichnet werden. Wenn der Wert unter der Minimumgrenze ist, dann wird es als Minimum interpretiert.
Die anderen Werte werden als undefiniert bezeichnet.
So entsteht ein langer Strang von Maximum, Minimum und undefinierten Werten.
Nun muss nach der Folge Minimum, Maximum oder Maximum, Minimum gesucht werden. Für die Folgen werden dann die entsprechenden Bits gesetzt.
Nach einer langen Folge von undefinierten Werten, darf man davon ausgehen, dass hier eine Pause vorliegt.
Wenn man nun immer 8 Bits aneinander reiht und diese als Byte bzw. als Char interpretiert, dann bekommt man die Nachricht:

```
Dies ist ein geheimes Gedicht,
bekannt werden will's niemals nicht.
Doch bist du ganz schlau
und auch noch genau,
```

```
bringst du seinen Inhalt ans Licht/
```

2.2. Programm-Dokumentation

Die Klasse *Manager* ist die Hauptklasse. Sie übernimmt alle Aufgaben.

Um nun die Nachricht zu bekommt, muss zuerst ein Objekt der Klasse *Manager* erstellt werden. Dann läd man mit der überladenen Methode *Load(string), Load(Stream)* die Wavedatei entweder aus einer Datei durch Angabe des Dateinamens oder aus einem Stream.
Beim Laden werden die beiden Arrays *byteArray* und *shortArray* gefüllt.

Danach Ruft man die Methode *RemoveNoise(int)* auf. Sie lässt intern eine For-Schleife durchlaufen und ruft dabei die private Methode *RemoveNoise()* auf. Um eine gute Rauschverminderung zu erhalten, sollte der Parameter ca. 50 sein.

Nun kann man die Methode *GetMessage(int) : string* aufrufen. Der Parameter beschreibt, wie groß die Pause, die die Töne voneinander trennt ungefähr ist.

Die beiden Methoden *GetByteFormBits(byte[])* und
GetBitsFromByte(byte b) sind dafür da, um 8 Bits in ein Byte oder ein Byte in 8 Bits umzuwandeln. Eine Erweiterung ist, dass man selbst eigene Texte nach dem Prinzip verschlüsseln kann.

Intern wurde schon in den Ressourcen, die man bei Visual Studio erstellen kann, 2 Dateien geladen: Bit0.wav und Bit1.wav.

Wenn man die Methode *CreateSound(string, string, int)* aufruft, muss man im 1. Parameter den Text angeben, der verschlüsselt werden soll. Im 2. Parameter den Zieldateinamen, und im 3. Parameter wie lange eine Pause zwischen den Tönen sein soll.
Intern wird die Methode *CreateWaveHeader()* aufgerufen.

2.3. Programm-Ablaufprotokolle

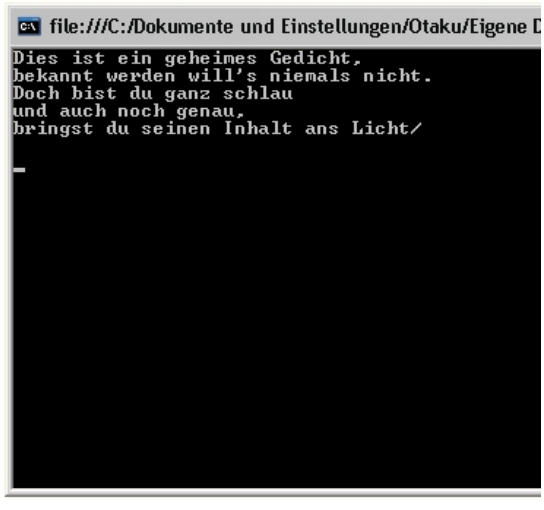
```
namespace Geheimnisvolle_toene
{
    class Program
    {
        [STAThread]
        static void Main(string[] args)
        {
            Manager man = new Manager();

            man.Load("original.wav");

            man.RemoveNoise(50);

            Console.WriteLine(man.GetMessage(100));

            Thread.Sleep(5000);
        }
    }
}
```



In der ersten Abbildung sieht man, wie die Datei „original.wav“ geladen, das Rauschen entfernt und der Text entschlüsselt wird.
Der Text wird dann in die Konsole geschrieben.

```
    {
        [STAThread]
        static void Main(string[] args)
        {
            Manager manCreate = new Manager();

            manCreate.CreateSound("Die ist ein anderes Gedicht\ndoch von dem BWINF nicht."
                                  ,"output.wav", 100);


            Manager manLoader = new Manager();

            manLoader.Load("output.wav");

            Console.WriteLine(manLoader.GetMessage(50));

            Thread.Sleep(10000);
        }
    }
}
```

In der zweiten Abbildung wird eine Datei mit dem Namen „output.wav“ erstellt. In ihr wird die Nachricht „Die ist ein anderes Gedicht\ndoch von dem BWINF nicht.“ gespeichert.
Mit dem Objekt *manLoader* wird die selbe Datei geladen und der Text entschlüsselt und in die Konsole geschrieben.

2.4. Programm-Text

```
using System;
using System.Collections.Generic;
using System.IO;


namespace Geheimnisvolle_toene
{
public enum Maximum
{
                Max,
                Min,
                Ka
}

class Manager
{
                private byte[] byteArray;
                private short[] shortArray;

                public Manager()
                {
```

```
}

public Manager(short[] shortArray)
{
   this.shortArray = shortArray;
   CreateWaveHeader();
   SetShortToByteArray();
}

private void CreateWaveHeader()
{
   byteArray = new byte[44 + shortArray.Length * 2];

   MemoryStream ms = new MemoryStream(byteArray);

   //0          4          'RIFF'
   //4          4          <Dateigröße - 8>
   //8          4          'WAVE'

   //12         4          'fmt '          Header Signatur
   //16         4          <fmt length>    Länge des restlichen fmt-Headers (16 Byte)

   //20         2          <format tag>    sample Datenformat (siehe separate Tabelle weiter unten)
   //22         2          <channels>      Anzahl der Kanäle: 1 = mono, 2 = stereo
   //24         4          <sample rate>               Abtastrate pro Sekunde (z.B. 44100)
   //28         4          <bytes/second>              Sample-Rate * Block-Align
   //32         2          <block align>Kanäle * bits/sample / 8
   //34         2          <bits/sample>               8, 16 oder 24

   //36         4          'data'          Header Signatur
   //40         4          <length>        Länge des Datenblocks
   //44         <bits/sample>/8                        Sample Data

   ms.WriteByte((byte)'R');
   ms.WriteByte((byte)'I');
   ms.WriteByte((byte)'F');
   ms.WriteByte((byte)'F');

   ms.Write(BitConverter.GetBytes(byteArray.Length), 0, 4);

   ms.WriteByte((byte)'W');
   ms.WriteByte((byte)'A');
   ms.WriteByte((byte)'V');
   ms.WriteByte((byte)'E');

   ms.WriteByte((byte)'f');
   ms.WriteByte((byte)'m');
   ms.WriteByte((byte)'t');
   ms.WriteByte((byte)' ');

   ms.Write(BitConverter.GetBytes((int)16), 0, 4);


   ms.Write(BitConverter.GetBytes((short)1), 0, 2);

   ms.Write(BitConverter.GetBytes((short)1), 0, 2);

   ms.Write(BitConverter.GetBytes((int)44100), 0, 4);
```

```
        ms.Write(BitConverter.GetBytes((int)(44100 * 1 * (16 / 8))), 0, 4);

        ms.Write(BitConverter.GetBytes((short)1 * (16 / 8)), 0, 2);

        ms.Write(BitConverter.GetBytes((short)16), 0, 2);

        ms.WriteByte((byte)'d');
        ms.WriteByte((byte)'a');
        ms.WriteByte((byte)'t');
        ms.WriteByte((byte)'a');

        ms.Write(BitConverter.GetBytes((int)shortArray.Length * 2), 0, 4);

        ms.Close();
    }

    //EXPORT - IMPORT

    public void Load(string filename)
    {
        FileStream fs = new FileStream(filename, FileMode.Open);
        byteArray = new byte[(int)fs.Length];
        fs.Read(byteArray, 0, (int)fs.Length);
        fs.Close();

        List<short> shortList = new List<short>();
        for (int i = 44; i < byteArray.Length - 44; i += 2)
        {
            shortList.Add(BitConverter.ToInt16(new byte[]{ byteArray[i],byteArray[i + 1] }, 0));
        }
        shortArray = shortList.ToArray();
    }

    public void Load(Stream s)
    {
        byteArray = new byte[(int)s.Length];
        s.Read(byteArray, 0, (int)s.Length);

        List<short> shortList = new List<short>();
        for (int i = 44; i < byteArray.Length - 44; i += 2)
        {
            shortList.Add(BitConverter.ToInt16(new byte[] { byteArray[i], byteArray[i + 1] }, 0));
        }
        shortArray = shortList.ToArray();
    }

    public void Save(string filename)
    {
        FileStream fs = new FileStream(filename, FileMode.Create);
        fs.Write(byteArray, 0, byteArray.Length);
        fs.Close();
    }

    //NOISE

    public void RemoveNoise(int often)
    {
```

```
    for (int i = 0; i < often; i++)
    {
      RemoveNoise();
    }

    SetShortToByteArray();
  }

  private void RemoveNoise()
  {
    int toleranz = 50;

    if (shortArray.Length > 0)
    {
      for (int i = 0; i < shortArray.Length - 1; i++)
      {
        if (shortArray[i] > shortArray[i + 1] + toleranz)
        {
          short dif = (short)(shortArray[i] - shortArray[i + 1]);

          shortArray[i] -= (short)(dif / 2);
          shortArray[i + 1] += (short)(dif / 2);
        }
        else if (shortArray[i] < shortArray[i + 1] - toleranz)
        {
          short dif = (short)(shortArray[i + 1] - shortArray[i]);

          shortArray[i + 1] -= (short)(dif / 2);
          shortArray[i] += (short)(dif / 2);
        }
      }
    }
  }

  private void SetShortToByteArray()
  {
    if (shortArray.Length > 0)
    {
      byte[] bytes;

      for (int i = 0; i < shortArray.Length; i++)
      {
        bytes = BitConverter.GetBytes(shortArray[i]);

        byteArray[(2 * i) + 44] = bytes[0];
        byteArray[(2 * i) + 44 + 1] = bytes[1];
      }
    }
  }

  //Add

  public void AddNoise(int minLvl, int maxLvl)
  {
    Random rand = new Random();

    for (int i = 0; i < shortArray.Length; i++)
    {
```

```
            int UpOrDown = rand.Next(0, 2);

            int change = rand.Next(minLvl, maxLvl);

            if (UpOrDown == 0)
            {
                shortArray[i] += (short)(change * 100);
            }
            else
            {
                shortArray[i] -= (short)(change * 100);
            }
        }

        SetShortToByteArray();
    }

    //GET MESSAGE

    public string GetMessage(int pauseLength)
    {
        bool steigt = false;
        bool steigtBefor = false;
        List<int> intList = new List<int>();

        //EXTREMAS HOLEN
        for (int i = 0; i < shortArray.Length - 1; i += 2)
        {
            if (shortArray[i] > shortArray[i + 1])
            {
                steigt = false;
            }
            else if (shortArray[i] < shortArray[i + 1])
            {
                steigt = true;
            }

            if (steigt != steigtBefor)//richtungswechel
            {
                intList.Add(shortArray[i]);

                steigtBefor = steigt;
            }
        }


        //alles was über maxFrontier liegt ist maximal
        //alles was unter minFrontier liegt ist minimal
        int maxFrontier = short.MaxValue / 2;
        int minFrontier = short.MinValue / 2;
        List<Maximum> extremaList = new List<Maximum>();
        for (int i = 0; i < intList.Count; i++)
        {
            if (intList[i] > maxFrontier)
            {
                extremaList.Add(Maximum.Max);
            }
            else if (intList[i] < minFrontier)
```

```
    {
        extremaList.Add(Maximum.Min);
    }
    else
    {
        extremaList.Add(Maximum.Ka);
    }
}

//Max Min = 0
//Min Max = 1
//Ka Ka Ka Ka Ka = pause
List<byte> bitList = new List<byte>();
List<byte[]> fertigeArraysList = new List<byte[]>();
int KaCount = 0;
int KaFrontier = pauseLength;//ka pause zwischen den bits
bool warteAufPause = false;
string message = "";
byte[] bits = new byte[8];

for (int i = 0; i < extremaList.Count - 1; i += 1)
{
    if (bitList.Count == 8)
    {
        message += (char)GetByteFormBits(bitList.ToArray());
        bitList.Clear();
    }

    if (extremaList[i] == Maximum.Max && extremaList[i + 1] == Maximum.Min)
    {
        if (!warteAufPause)
        {
            bitList.Add(1);
            warteAufPause = true;
            KaCount = 0;
            continue;
        }
    }
    else if (extremaList[i] == Maximum.Min && extremaList[i + 1] == Maximum.Max)
    {
        if (!warteAufPause)
        {
            bitList.Add(0);
            warteAufPause = true;
            KaCount = 0;
            continue;
        }
    }
    else
    {
        KaCount++;
    }

    //PAUSE
    if (KaCount > KaFrontier)
    {
        warteAufPause = false;
        KaCount = 0;
```

```
            }
        }

        return message;
    }

    //CREATE MESSAGE

    public void CreateSound(string message, string filename, int littlePauseLenght)
    {
        int bigPauseLenght = 1000;

        Random rnd = new Random();

        Manager ton0 = new Manager();
        //ton0.Load(soundRes._0___Ton);
        ton0.Load(soundRes.bit0);

        Manager ton1 = new Manager();
        //ton1.Load(soundRes._1___Ton);
        ton1.Load(soundRes.bit1);

        List<short> shortList = new List<short>();

        for (int i = 0; i < message.Length; i++)
        {
            byte[] bits = GetBitsFromByte((byte)message[i]);

            for (int j = 0; j < bits.Length; j++)
            {
                if (bits[j] == 0)
                {
                    for(int k = 0; k < ton0.ShortArray.Length; k++)
                    {
                        shortList.Add(ton0.ShortArray[k]);
                    }

                    for (int k = 0; k < ton0.ShortArray.Length; k++)
                    {
                        shortList.Add(ton0.ShortArray[k]);
                    }

                    for (int k = 0; k < ton0.ShortArray.Length; k++)
                    {
                        shortList.Add(ton0.ShortArray[k]);
                    }
                }
                else if (bits[j] == 1)
                {
                    for (int k = 0; k < ton1.ShortArray.Length; k++)
                    {
                        shortList.Add(ton1.ShortArray[k]);
                    }

                    for (int k = 0; k < ton1.ShortArray.Length; k++)
                    {
                        shortList.Add(ton1.ShortArray[k]);
                    }
```

```
                for (int k = 0; k < ton1.ShortArray.Length; k++)
                {
                    shortList.Add(ton1.ShortArray[k]);
                }
            }

            //längere pause einfügen
            for (int k = 0; k < bigPauseLenght; k++)
            {
                shortList.Add((short)rnd.Next(0,100));
            }
        }

        //längere pause einfügen
        for (int k = 0; k < bigPauseLenght; k++)
        {
            shortList.Add((short)rnd.Next(0, 100));
        }
    }

    Manager output = new Manager(shortList.ToArray());

    output.Save(filename);
}

//Tools

private byte GetByteFormBits(byte[] bits)
{
    int value = 0;

    if (bits.Length == 8)
    {
        value += bits[7] * (int)Math.Pow(2, 0);
        value += bits[6] * (int)Math.Pow(2, 1);
        value += bits[5] * (int)Math.Pow(2, 2);
        value += bits[4] * (int)Math.Pow(2, 3);
        value += bits[3] * (int)Math.Pow(2, 4);
        value += bits[2] * (int)Math.Pow(2, 5);
        value += bits[1] * (int)Math.Pow(2, 6);
        value += bits[0] * (int)Math.Pow(2, 7);

        return (byte)value;
    }

    return 0;
}

private byte[] GetBitsFromByte(byte b)
{
    byte[] bits = new byte[8];

    int byteInt = ((int)b);

    for (int i = 0; i < 8; i++)
    {
        if (byteInt >= Math.Pow(2, 7 - i))
```

```
                {
                    bits[i] = 1;
                    byteInt -= (int)Math.Pow(2, 7 - i);
                }
                else
                {
                    bits[i] = 0;
                }
            }


            return bits;
        }

        //proberties

        public short[] ShortArray
        {
                        get
                        {
                            return shortArray;
                        }
        }
}
}
```