32. Bundeswettbewerb Informatik, 1. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

Das Wichtigste zuerst: Wir haben uns sehr darüber gefreut, dass wieder besonders viele sich die Mühe gemacht und die Zeit zur Bearbeitung der Aufgaben genommen haben! Natürlich waren nicht alle Einsendungen perfekt, und einige eher äußerliche Anforderungen wurden häufiger missachtet. Im Einzelnen:

- Die Dokumentationen zu den Aufgaben, also das bzw. die in der Einsendung enthaltenen PDF-Dokument(e), werden für die Bewertung ausgedruckt. Aus Zeitgründen kann es sein, dass die Bewertung nur auf der Grundlage der ausgedruckten Unterlagen erfolgt. Die Beschreibungen von Lösungsidee und Umsetzung, insbesondere aber auch ausreichend viele Beispiele und der Quelltext (bis auf unwichtige Teile) müssen deshalb in der Dokumentation enthalten sein. Aus Zeit- und Kostengründen ist es unmöglich, alle Einsendungen nach weiterem Material zu durchsuchen und dieses auszudrucken.
- Das Fehlen von Beispielen, erst recht von vorgegebenen Beispielen, führt zu Punktabzug. Es ist nicht ausreichend, Beispiele nur in gesonderten Dateien abzugeben, ins Programm einzubauen oder den Bewertern das Erfinden und Testen von Beispielen zu überlassen. Leider fehlten in vielen Einsendungen die Beispiele, was oft das Erreichen der zweiten Runde verhindert hat.
- Zu einer Einsendung gehören lauffähige Programme, ohne ist die Bewertung schwierig. Kompilierung von Quellcode ist während der Bewertung nicht möglich. Für die gängigsten Skript-Sprachen stehen Interpreter zur Verfügung. Einige Entwicklungsumgebungen (z. B. BlueJ), bei denen die erstellten Programme ohne Weiteres nur in der Umgebung selbst laufen, stehen bei der Bewertung zur Verfügung, aber sicher nicht alle.

Vielleicht helfen diese Anmerkungen, wenn du (hoffentlich) im nächsten Jahr wieder mitmachst. Auch die folgenden eher inhaltlichen Dinge sind zu beachten:

- Lösungsideen sollten keine Bedienungsanleitungen oder Wiederholungen der Aufgabenstellung sein. Es soll beschrieben werden, welches Problem hinter der Aufgabe steckt und wie dieses Problem grundsätzlich angegangen wird. Ein einfacher Grundsatz: Bezeichner von Programmelementen wie Variablen, Prozeduren etc. werden nicht verwendet. Eine Lösungsidee ist nämlich unabhängig von solchen Realisierungsdetails.
- Die Beispiele sollen die Korrektheit der Lösung belegen. Es sollten auch Sonderfälle gezeigt werden, die die Lösung behandeln kann. Die Konstruktion solcher Testfälle ist eine ganz wesentliche Tätigkeit des Programmentwurfs.

Einige Anmerkungen noch zur Bewertung:

- Pro Aufgabe werden maximal fünf Punkte vergeben. Für die Gesamtbewertung sind die drei am besten bewerteten Aufgabenlösungen maßgeblich; es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreicht man mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten und eine Anerkennung mit 9 bis 11 Punkten. Die Preisträger sind für die zweite Runde qualifiziert.
- In der Juniorliga wird ein 1. Preis für 9 oder 10 Punkte, ein 2. Preis für 7 oder 8 Punkte und eine Anerkennung für 5 oder 6 Punkte vergeben. Leider gibt es in der Juniorliga (noch) keine zweite Runde.
- Eine Einsendung wird in Juniorliga und Hauptliga gewertet, wenn alle Gruppenmitglieder die Altersbedingung für Junioraufgaben erfüllen (damit kommt die Einsendung für die Juniorliga in Frage) und in der Einsendung sowohl Junioraufgaben als auch andere Aufgaben bearbeitet wurden.
- Auf den Bewertungsbögen bedeutet ein Kreuz in einer Zeile, dass die (meist negative) Aussage in dieser Zeile auf die Einsendung zutrifft. Damit verbunden ist dann in der Regel der Abzug eines oder mehrerer Punkte. Eine Wellenlinie ($\sim$) bedeutet „na ja, hätte besser sein können“, führt aber alleine nicht zu Punktabzug. Mehrere Wellenlinien können sich aber zu einem Punktabzug addieren. Gelegentlich sind lobende Anmerkungen der Bewerter mit einem ’+’ versehen.
- Leider ließ sich nicht verhindern, dass etliche Teilnehmer nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden. Das ist ziemlich riskant, da Fehler sich leicht einschleichen.

Zum Schluss:

- Sollte der Name auf der Urkunde falsch geschrieben sein, kann gerne eine neue angefordert werden.
- Es ist verständlich, wenn jemand, der nicht weitergekommen ist, über eine Reklamation nachdenkt. Kritische Fälle, insbesondere die mit 11 Punkten, haben wir allerdings schon gründlich und mit Wohlwollen geprüft.

Danksagung

An der Erstellung der Lösungsideen haben mitgewirkt: Friedrich Hübner (Junioraufgabe 1), Thomas Leineweber (Junioraufgabe 2), Martin Thoma (Aufgabe 1), Nikolai Wyderka (Aufgabe 2), Matthias Kemper (Aufgabe 3) und Philip Wellnitz (Aufgabe 5).

Die Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt, und zwar aus Vorschlägen von Wolfgang Pohl (Junioraufgabe 1), Torben Hagerup (Junioraufgabe 2, Aufgabe 1 und Aufgabe 2), Jens Gallenbacher (Aufgabe 3), Peter Rossmanith (Aufgabe 4) und Ralf Punkenburg (Aufgabe 5).

Junioraufgabe 1: Songwriter

J1.1 Lösungsidee

Von der Aufgabenstellung ausgehend bietet es sich an, den Songtext zufällig zu bestimmen. Das ermöglicht eine größere Breite an ausgegebenen Liedtexten. In den folgenden Schritten wird deswegen mehrmals ein Zufallsgenerator verwendet. Dies ist aber nicht notwendig. Insbesondere können Daten als Eingabe erwartet werden. In diesem Fall muss aber ein bestimmtes Eingabeformat definiert werden.

Ermitteln eines zufälligen Songtextes

Ein Lied ist aus Strophen, eine Strophe aus mehreren Zeilen und eine Zeile aus Silben aufgebaut. Eine Silbe ist ein Konsonant und ein Vokal.

```
function SILBE( )
    return (zufälliger Konsonant)+(zufälliger Vokal)
```

Um eine Zeile zu erzeugen, müssen mehrere solcher Silben zusammengefügt werden. In der Mitte soll ein 'p di' eingefügt werden[1]:

```
function ZEILE(anzahlsilben)
    silbe ← SILBE( )
    zeile ← ' '
    for i ← 1 to anzahlsilben do
        zeile ← zeile + silbe
        if i = (anzahlsilben − 1)/2 then
            zeile ← zeile + 'p di'
    return zeile
```

Eine Strophe besteht aus mehreren Zeilen. Diese sollten alle die gleiche Silbenzahl haben. Durchschnittlich jede zweite Strophe wird durch „pseudo-cooles Zeugs“ ergänzt.

```
function STROPHE(anzahlzeilen, silbenzahl)
    EINSCHRÄNKEN( )
    strophe ← ' '
    for i ← 1 to anzahlzeilen do
        strophe ← strophe + ZEILE(silbenzahl)
    if random(0,1) < 0.5 then
        strophe ← strophe + (pseudo-cooles Zeugs)
    return strophe
```

[1] Hinweis: In den Pseudocode-Beispielen wurde der Übersichtlichkeit wegen darauf verzichtet, Leerzeichen und Zeilenumbrüche aufzuführen

Einschränken der verwendeten Buchstaben

Die Prozedur EINSCHRÄNKEN schränkt die Vokale und Konsonanten ein, die für die Strophe verwendet werden können. Die Anzahl der Vokale soll möglichst klein sein. Es ist aber zu beachten, dass mindestens ein Vokal und ein Konsonant verwendet werden müssen. Deswegen werden die Vokale getrennt von den Konsonanten eingeschränkt. Im Folgenden wird eine Lösung vorgestellt, wie man diese Bedingung mit Hilfe eines Zufallsgenerators erfüllen kann. An dieser Stelle gibt es viele Möglichkeiten, die Bedingung 'Kleine Anzahl' zu interpretieren. Anstatt jeder möglichen Anzahl an Vokalen die gleiche Wahrscheinlichkeit zuzuordnen, kann man für eine höhere Wahrscheinlichkeit für weniger Elemente sorgen. Dazu bietet sich eine Logarithmus- oder eine Wurzelfunktion an:

procedure EINSCHRÄNKEN( )
  anzahlvokale $\leftarrow 5 - \lfloor\sqrt{random[0,5^2)}\rfloor$
  verfügbarevokale ← Menge mit (anzahlvokale) versch. Vokalen
  ...wiederhole für die Konsonanten

Die Zahl 5 wird als Parameter verwendet, da es 5 Vokale (a,e,i,o,u) gibt. Möchte man die Konsonanten einschränken, muss man entsprechend die Anzahl der Konsonanten einsetzten. Die Funktion $5 - \lfloor\sqrt{random[0,5^2)}\rfloor$ ($\lfloor\,\rfloor$ bedeutet „abrunden“) gibt mit unterschiedlicher Wahrscheinlichkeit eine Zahl von 1 bis 5 zurück:

Tabelle 1: Wahrscheinlichkeit der Ergebnisse

Ergebnis	mögl. Zufallszahlen	Wahrscheinlichkeit
5	0	1/25
4	1, 2, 3	3/25
3	4, 5, 6, 7, 8	5/25
2	9, 10, 11, 12, 13, 14, 15	7/25
1	16, 17, 18, 19, 20, 21, 22, 23, 24	9/25

Gesamtes Lied erstellen

Um aus den Strophen ein Lied zu erstellen, muss man mehrere Strophen zusammenfügen. Allerdings soll die Zeilenanzahl der Strophen ein Muster haben. Es gibt viele Möglichkeiten, dieses Muster zu erzeugen. Am abstraktesten ist es, eine Funktion $f(n)$ zu definieren. Diese Funktion liefert zu einer Strophe n die Zeilenanzahl $f(n)$. Sollen z. B. in jeder Strophe vier Zeilen sein, so definiert man $f(n) = 4$. Eine ähnliche Funktion $g(n)$ gibt die Silbenzahl der Strophe zurück.

function LIED
  anzahlstrophen ← random()
  lied ← ' '
  for i ← 1 to anzahlstrophen **do**
    lied ← lied + STROPHE($(f(i), g(i))$)
  return lied

J1.2 Umsetzung

Das Programm kann im Grunde genommen wie oben beschrieben aufgebaut werden. Die meisten Funktionen können mit entsprechender Syntax in den Quellcode übernommen werden.

Allgemeine Hinweise

- Das Verwenden eines Zufallsgenerators ist nicht zwingend notwendig. Wird er verwendet, sollten Maximalwerte für die Silbenanzahl, Zeilenanzahl und die Strophenanzahl festgelegt werden. Zudem muss man beachten, dass in der Aufgabenstellung Minimalwerte verlangt sind (z. B. mindestens 2 Strophen).
- Es ist legitim, bei den Vokalen die Umlaute äöü mit einzubeziehen. Bei den Konsonanten kann man sich überlegen, 'q' auszuschließen, da 'q' immer in Verbindung mit 'u' steht.

Im Folgenden werden noch einige Hinweise zu bestimmten Funktionen gegeben.

EINSCHRÄNKEN() Auf irgendeine Art muss die Anzahl der verwendbaren Buchstaben heruntergesetzt werden. Die genaue Anzahl kann z. B. wie in der Lösungsidee vorgeschlagen, zufällig bestimmt worden sein. Das Einschränken vereinfacht sich erheblich, wenn man ein Array von Buchstaben definiert, in dem alle verwendbaren Vokale und Konsonanten enthalten sind (z. B. Java: char[ ] vokale = {'a', 'e', 'i', 'o', 'u'}). Um die Buchstaben einzuschränken, wählt man eine Teilmenge dieser Menge an Buchstaben aus. Wieder bietet es sich an, eine Teilmenge der Buchstaben zufällig zu bestimmen. Dieses Problem kann auf viele Arten gelöst werden. Die meisten Varianten sind jedoch nicht einfach zu programmieren und werden schnell unübersichtlich. Eine relativ einfache Variante ist, die Elemente des Arrays zufällig anzuordnen. Der Algorithmus dafür ist in der Regel in Standardbibliotheken der gängigen Programmiersprachen implementiert (z. B. C++: std::random_shuffle, Java: Collections.shuffle, Python: random.shuffle). Hat man eine zufällige Permutation, kann man die ersten n Elemente des Arrays zurückgeben.

LIED() Um die Funktion LIED() zu implementieren, muss man sich zuerst Gedanken über die Implementierung der Zeilenzahl- bzw. Silbenzahlfunktionen $f(n)$ und $g(n)$ machen. Man kann entweder ein Muster festlegen oder zwischen mehreren zufällig wählen. Benutzt man die zweite Variante, so bestimmt man zu Beginn der Funktion eine zufällige Zahl. Erstellt man die einzelnen Strophen, so benutzt man If-Klauseln bzw. Switch-Case-Klauseln um zu einer Strophe die Zeilenanzahl zu bestimmen.

J1.3 Beispiel

```
KA KA KA KAP DI KA KA KA
BI BI BI BIP DI BI BI BI
YE YE YE YEP DI YE YE YE
TE TE TE TEP DI TE TE TE
FAKE THAT!

DAP DI
KAP DI
DAP DI
YAP DI
YO MAN

RE RE RE REP DI RE RE RE
RE RE RE REP DI RE RE RE
RE RE RE REP DI RE RE RE
RO RO RO ROP DI RO RO RO
RE RE RE REP DI RE RE RE
RO RO RO ROP DI RO RO RO
YEAH!

SUP DI
SEP DI
SEP DI
GEP DI
LEP DI
GEP DI
```

J1.4 Bewertungskriterien

- Die ausgegebenen Songs sollen natürlich den grundlegenden Regeln aus der Aufgabenstellung entsprechen. Insbesondere war zu beachten:
  - die Einschränkung der für eine Strophe zur Verfügung stehenden Buchstabenmenge;
  - die ungerade Anzahl von Grundsilben in einer Zeile, mit 'p di' an der mittleren Silbe; und
  - die Silbenzahl einer Strophe (alle Zeilen einer Strophe haben die gleiche Anzahl von Silben).
- Die Muster für Zeilenzahlen und Silbenzahlen sollen geeignet realisiert werden.
- Eine weitere Anforderung der Aufgabe ist, dass ausgegebene „Songtexte“ deutlich variieren. Das geht am einfachsten mit Zufallszahlen, aber andere Wege sind auch erlaubt.
- Mindestens drei unterschiedliche Songtexte sind als Beispiele gefordert. Aber: die werden vermutlich auch bei guten Lösungen nicht immer vorhanden sein. Bitte in der Einsendung schauen und ggf. auch Programm laufen lassen.

Junioraufgabe 2: Zollstock

J2.1 Lösungsidee

Um herauszufinden, ob sich schon eine Faltung des Zollstocks wiederholt haben muss, ist es notwendig, zuerst die *Anzahl der möglichen Faltungen* zu bestimmen. Der Zollstock besteht aus zehn Segmenten, die mit neun Gelenken miteinander verbunden sind. Dabei kann jedes Gelenk entweder nicht gefaltet oder gefaltet sein. Der Zollstock hat an einem Ende ein Loch. Damit ist z. B. die Faltung, bei der nur das erste Gelenk neben dem Lochsegment gefaltet ist, unterscheidbar von der Faltung, bei der nur das letzte Gelenk gefaltet ist. Nicht unterscheidbar ist, wenn der Zollstock gedreht wird oder ob eine Faltung links rum oder rechts rum durchgeführt wurde. Damit gibt es insgesamt $2^9 = 512$ mögliche Faltungen.

Eine Faltung des Zollstocks kann als Zeichenkette der Länge 9 beschrieben werden, in der für jedes gefaltete Gelenk eine **1**, für jedes nicht gefaltete Gelenk eine **0** steht. Die zwei oben erwähnten Faltungen werden dann durch **100000000** beziehungsweise **000000001** beschrieben.

Aber nicht alle 512 möglichen Faltungen sind auch *gültige Faltungen*. Eine Faltung ist eine gültige Faltung, wenn der gefaltete Zollstock auch in den Stoffbeutel passt. Als Mindestbedingung gilt, dass zwischen zwei gefalteten Gelenken maximal ein nicht gefaltetes Gelenk sein darf. Andernfalls wären drei Segmente in Verlängerung zueinander, und der Zollstock passt nicht mehr hinein. Auf die Darstellung als Zeichenkette übertragen, bedeutet es, dass in einer gültigen Faltung die **0** nicht zweimal direkt nebeneinander stehen darf. Leider reicht diese Bedingung nicht aus. So sind z. B. in der Faltung **011011011** nie drei Segmente direkt in einer Reihe. Der gefaltete Zollstock hat aber eine Länge von 80 Zentimetern.

Da es nur 512 insgesamt mögliche Faltungen gibt, kann man alle möglichen Faltungen erzeugen und prüfen, wie viele der Faltungen gültige Faltungen sind. In zwei Monaten gibt es circa 40 Arbeitstage[2]. Wenn es nun weniger gültige Faltungen gibt, muss sich eine Faltung in der Zeit zwingend wiederholt haben. Wenn es mehr gültige Faltungen gibt, dann kann es sein, dass sich noch keine Faltung wiederholt hat.

J2.2 Umsetzung

Für die Umsetzung generiert man als Zeichenkette alle möglichen Faltungen und prüft für jede Faltung, ob sie auch gültig ist. Die gültigen Faltungen werden gezählt. Da es nur 512 verschiedene Faltungen gibt, ist dies schnell und einfach machbar.

Die Überprüfung, ob eine Faltung gültig ist, findet wie folgt statt:

```
private static boolean istGueltig(String stab) {
  int pos = 1; //Endposition des ersten Teiles
  int min = 0; //Unteres Ende des gefalteten Stabes
  int max = 1; //oberes Ende des gefalteten Stabes
  boolean richtung = true; // das letzte Teil zeigte nach oben
```

[2] Statt 40 kann auch eine andere plausible Zahl an Arbeitstagen als Vergleichszahl gewählt werden.

```
    //Gehe nun alle Gelenke durch
    for (int i = 0; i < stab.length(); i++) {
      char gelenk = stab.charAt(i);
      if (gelenk == '1') {
        //Bei einer Faltung die Richtung umkehren
        richtung = !richtung;
      }

      //Bestimme Endposition des nächsten Teiles
      if (richtung) {
        pos++;
      } else {
        pos--;
      }

      //Gibt es neue absolute Grenzen des Stabes?
      if (pos<min) min=pos;
      if (pos>max) max=pos;
    }

    //Wenn der Abstand zwischen den beiden Grenzen >2 ist,
    //dann ist die Faltung nicht gültig
    if (max - min > 2)
      return false;
    return true;
  }
```

Die Methode wird nun für jede mögliche Faltung aufgerufen, und es wird gezählt, wie oft das Ergebnis *true* ist.

J2.3 Beispiele

Ein mit der obigen Überprüfung arbeitendes Programm berechnet, dass von den 512 möglichen Faltungen nur 47 Faltungen gültig sind. Bei 40 Arbeitstagen in zwei Monaten ist es möglich, dass sich bisher noch keine Faltung wiederholt hat.

J2.4 Bewertungskriterien

Ein Programm zur Ermittlung der Anzahl der Faltungen war nicht gefordert, und daran haben sich die meisten Teilnehmer gehalten. Es sind also im Wesentlichen Überlegungen zur Ermittlung dieser Anzahl angestellt worden. Die Bewertungskriterien enthalten deshalb keine Ansprüche an evtl. realisierte Programme: damit diejenigen, die sich die Mühe des Programmierens gemacht haben, für eventuelle Mängel des Programms nicht bestraft werden.

- Die Vorgehensweise zur Ermittlung der Faltungsanzahl soll gut nachvollziehbar beschrieben sein. Beispielhafte Abbildungen bzw. Illustrationen sind hier sehr willkommen.
- Bei einem falschen Gültigkeitskriterium, einem symmetrischen Zollstock oder einem anderweitig falschen Ergebnis gibt es Abzug. Ein typischer Fehler ist, die Länge, die der gefaltete Zollstock schließlich hat, nicht korrekt zu beachten. Häufig wurde auch die durch das Loch zum Aufhängen gegebene Orientierung des Zollstocks übersehen.
- Es muss eine Aussage dazu gemacht werden, ob sich schon eine Faltung in den zwei Monaten wiederholt haben kann. Die Aussage muss auf Grund der Anzahl der vorher bestimmten gültigen Faltungen nachvollziehbar sein. Die Wahl der Vergleichszahl wird typischerweise zwischen 40 und 62 Tagen liegen; hier wird alles akzeptiert.
- Da sich dieses Problem nur mit einer Instanz beschäftigt (nämlich Ralucas Zollstock bzw. einem 2m-Zollstock mit 10 Segmenten), kann es keine weiteren Beispiele geben. Der übliche Bewertungspunkt zu „Beispielen“ entfällt hier also.

Aufgabe 1: Rationalisierung

Anna hat es erfolgreich geschafft, aus der dezimalen Darstellung einer rationalen Zahl x die Darstellung als Bruch $x = p/q$ zu gewinnen. Nun will Anna den Bruch vereinfachen, aber bei der Vereinfachung so nahe wie möglich an der Zahl x bleiben. Ein Bruch ist einfacher, wenn der Zähler oder der Nenner weniger Stellen haben.

Da das Vorzeichen bei der Vereinfachung keine Rolle spielt, gehen wir im Folgenden davon aus, dass $x > 0$ ist. Außerdem spielen die ganzzahligen Anteile keine Rolle, weil Anna eine Darstellung als gemischten Bruch wählen könnte. Also können wir auch von $x < 1$ ausgehen.

1.1 Kürzen mit Euklid

Durch Kürzen lässt sich ein Bruch vereinfachen, ohne Genauigkeit zu verlieren. Dazu teilt man sowohl Zähler als auch Nenner durch den größten gemeinsamen Teiler, den ggT. Der bekannteste Algorithmus zur Berechnung des ggT ist der euklidische Algorithmus. Dieser Algorithmus nutzt aus, dass für zwei ganze Zahlen $a, b \in \mathbb{N}_0$ mit $a \geq b$

$$\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$$

gilt.

Algorithmus 1 Euklidischer Algorithmus zur Berechnung des ggT

```
Eingabe: a,b ∈ ℕ0
  if a = 0 then
      return b
  while b ≠ 0 do
      if a > b then
          a ← a − b
      else
          b ← b − a
  return a
```

Sobald man sowohl Zähler p als auch Nenner q durch den $\mathrm{ggT}(p,q)$ geteilt hat, kann man nicht weiter kürzen. Diese vollständig gekürzte Darstellung eines Bruchs ist eindeutig. Das Kürzen ist immer eine Vereinfachung, daher sollte Anna niemals ein ungekürztes Ergebnis abgeben.

1.2 Brute-Force

Eine einfache Möglichkeit ein Ergebnis zu finden, das nahe am gegebenen Bruch $x = p/q$ ist, besteht darin, Zähler und Nenner versuchsweise zu ändern. Dazu addiert man Zahlen aus einem begrenzten Bereich zum Zähler und/oder zum Nenner und vergleicht das Ergebnis $\tilde{x}$ mit der exakten Zahl x. Algorithmus 2 ist eine Beschreibung dieses Verfahrens; beispielhaft

Algorithmus 2 Brute-Force I

Eingabe: $p, q \in \mathbb{N}_0$ mit $\text{ggT}(p,q) = 1$
$d \leftarrow \text{LÄNGE}(q)$
$p_{\text{opt}}, q_{\text{opt}} \leftarrow p, q$
for i in $-500, \ldots, 500$ **do**
  for j in $-500, \ldots, 500$ **do**
    $x \leftarrow \text{KÜRZEN}(\frac{p+i}{q+j})$
    if $\overbrace{\text{LÄNGE}(x.\text{nenner}) < d}^{\text{Bruch ist einfacher}}$ oder $\overbrace{\left(\text{LÄNGE}(x.\text{nenner}) = d \text{ and } |\frac{p}{q} - x| < |\frac{p}{q} - \frac{p_{\text{opt}}}{q_{\text{opt}}}|\right)}^{\text{Bruch ist exakter und gleich einfach}}$
  then
      $p_{\text{opt}}, q_{\text{opt}} \leftarrow x.\text{zähler}, x.\text{nenner}$
return $(p_{\text{opt}}, q_{\text{opt}})$

	Exakter Bruch x	Brute-Force-Bruch $\tilde{x}$	$\Delta = \lvert x - \tilde{x}\rvert$
1.857 143	$1857143/1000000$	$13/7$	0.000 000 14
1.293 982	$646991/500000$	$22/17$	0.000 135 65
3.141 593	$3141593/1000000$	$22/7$	0.001 264 14
1.414 214	$707107/500000$	$58/41$	0.000 420 15

Tabelle 2: Ergebnisse des Algorithmus Brute-Force I

wird der Bereich $[-500, 500]$ für die „Änderungszahlen“ verwendet. Damit erhält man die Ergebnisse aus Tabelle 2.

Erlaubt man einen maximal n-stelligen Nenner, dann gibt es immer einen Bruch $\tilde{x} = \tilde{p}/\tilde{q}$, der weniger als $\frac{1}{\underbrace{9\ldots9}_{n \text{ mal } 9}} = \frac{1}{10^n-1}$ von einem gegebenen Bruch $x = p/q$ entfernt ist. Da davon auszugehen ist, dass Annas Lehrer nur Brüche mit höchstens dreistelligen, eventuell noch vierstelligen Nennern akzeptiert, kann $\tilde{x}$ berechnet werden, indem alle möglichen Brüche zwischen 0 und 1 mit dem Nenner $\tilde{q} = 10^n - 1 < 10\,000$ betrachtet werden. Der Bruch $\tilde{x}$ mit dem geringstem Abstand $\Delta_{\min} = |x - \tilde{x}|$ ist der gesuchte Bruch. Er kann mit Algorithmus 3 ermittelt werden.

Algorithmus 3 Brute-Force II

Eingabe: $p, q, n \in \mathbb{N}_0$ mit $\text{ggT}(p,q) = 1$ und $p < q$
$\tilde{q} \leftarrow 10^n - 1$
$\Delta_{\min} \leftarrow x$
for i in $[1, \tilde{q} - 1]$ **do**
  $\tilde{x} \leftarrow \frac{i}{\tilde{q}}$
  if $|x - \tilde{x}| < \Delta_{\min}$ **then**
    $\tilde{p} \leftarrow i$
    $\Delta_{\min} \leftarrow |x - \tilde{x}|$
return $\tilde{p}, \tilde{q}$

Mit einem größeren n können wir eine bessere Garantie für die Genauigkeit angeben, bekommen aber auch einen Nenner mit n Stellen. Eventuell kann man den so gefundenen Bruch noch kürzen.

Algorithmus 3 und Kürzen liefert die Ergebnisse aus Tabelle 3.

	$n=2$		$n=3$		$n=4$	
x	$\tilde{x}$	$\Delta = \lvert x-\tilde{x}\rvert$	$\tilde{x}$	$\Delta = \lvert x-\tilde{x}\rvert$	$\tilde{x}$	$\Delta = \lvert x-\tilde{x}\rvert$
1.857 143	184/99	0.001 442 86	1855/999	0.000 286 14	6190/3333	0.000 042 72
1.293 982	128/99	0.001 052 71	431/333	0.000 312 29	4313/3333	0.000 047 40
3.141 593	311/99	0.000 178 86	1046/333	0.000 451 86	10471/3333	0.000 021 16
1.414 214	140/99	0.000 072 59	157/111	0.000 200 41	14141/9999	0.000 027 42

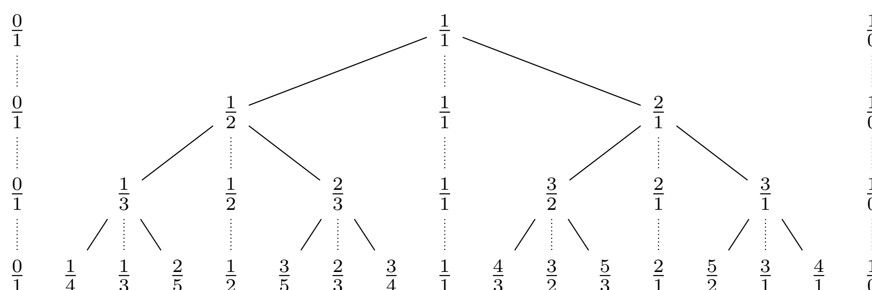
Tabelle 3: Ergebnisse des Algorithmus Brute-Force II

Interessant ist hier vor allem, dass 1.414 214 besser durch 113/99 approximiert wird als durch 157/111. Es bringt also nicht immer einen Vorteil, einen größeren Nenner zu wählen.

1.3 Stern-Brocot-Baum

Der Stern-Brocot-Baum ist ein binärer Baum mit unendlich vielen Knoten. Jeder Knoten repräsentiert einen vollständig gekürzten Bruch und jeder Bruch kommt genau ein mal im Baum vor. Die Wurzel des Baumes ist der Knoten 1/1 mit den Pseudoeltern 0/1 (links) und 1/0 (rechts).

Das linke Kind eines Knotens i/j mit dem linken Elternknoten l_z/l_n und dem rechten Elternknoten r_z/r_n ist $\frac{i+r_z}{j+r_n}$, das rechte Kind ist $\frac{i+l_z}{j+l_n}$:



Durch eine binäre Suche kann man nun systematisch Brüche finden, die nahe an der gesuchten Zahl sind. Dabei wird der aktuelle Knoten, beginnend bei der Wurzel 1, mit der Zahl verglichen. Ist die Zahl kleiner, steigt man im Baum links ab. Ist die Zahl größer, steigt man rechts ab. Da der Bruch, den ein Knoten repräsentiert, immer vollständig gekürzt ist, wird der Nenner größer oder bleibt gleich groß. Man kann also abbrechen, sobald der Nenner zu groß wird.

Es ist keineswegs so, dass die Näherung mit jedem Schritt besser wird. Wenn die Zahl beispielsweise 3/4 ist, wird die Näherung mit dem Abstieg von 1/1 auf 1/2 schlechter. Es ist also bei dem Abstieg nicht nur das Blatt, sondern der gesamte Pfad zu betrachten.

Es ergeben sich die Zahlen aus Tabelle 4.

x	$n=2$: $\tilde{x}$	$n=2$: $\Delta = \lvert x-\tilde{x} \rvert$	$n=3$: $\tilde{x}$	$n=3$: $\Delta = \lvert x-\tilde{x} \rvert$	$n=4$: $\tilde{x}$	$n=4$: $\Delta = \lvert x-\tilde{x} \rvert$
1.857143	13/7	0.00000014	13/7	0.00000014	13/7	0.00000014
1.293982	22/17	0.00013565	559/432	0.00000052	5053/3905	0.00000007
3.141593	311/99	0.00017886	355/113	0.00000008	355/113	0.00000008
1.414214	99/70	0.00007171	1393/985	0.00000080	1970/1393	0.00000007

Tabelle 4: Ergebnisse der Suche im Stern-Brocot-Baum mit maximal n-stelligem Nenner

1.4 Kettenbrüche

Ein Kettenbruch ist eine rationale Zahl

$$x = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cfrac{b_4}{\ddots}}}}$$

wobei alle a_i und alle b_i ganze Zahlen sind.

Eine wichtige Klasse von Kettenbrüchen sind reguläre Kettenbrüche; bei diesen sind alle $b_i = 1$. Für einen regulären Kettenbruch existiert eine kompaktere Schreibweise:

$$[a_0; a_1, a_2, a_3, \dots] := a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots}}}}$$

Zu jeder rationalen Zahl lässt sich mit dem so genannten „erweiterten Euklidischen Algorithmus“ (s. Algorithmus 4) ein regulärer Kettenbruch finden. Sei dazu $x = a/b$ eine Darstellung der rationalen Zahl x in vollständig gekürzter Form. Diese zerlegt man auf folgende Art:

$$\begin{aligned}
a &= a_0 b + r_1, && 0 < r_1 < b \\
b &= a_1 r_1 + r_2, && 0 < r_2 < r_1 \\
r_1 &= a_2 r_2 + r_3, && 0 < r_3 < r_2 \\
&\vdots && \vdots \\
r_{n-2} &= a_{n-1} r_{n-1} + r_n, && 0 < r_n < r_{n-1} \\
r_{n-1} &= a_n r_n + 0.
\end{aligned}$$

Der Kettenbruch ist dann $[a_0; a_1, a_2, \dots, a_{n-1}, a_n]$. Hierbei kann $a_0 < 0$ sein, aber alle anderen a_i sind größer als 0.

Damit ergibt sich der erweiterte Euklidische Algorithmus:

Algorithmus 4 Erweiterter Euklidischer Algorithmus

Eingabe: $a, b \in \mathbb{N}_0$ mit $\text{ggT}(a,b) = 1$
  $loesung \leftarrow \text{LIST}$
  $rest \leftarrow {}^{a}/{}_{b}$
  while $|rest| > 0$ **do**
    $rest \leftarrow a \mod b$
    $a_i \leftarrow \frac{a-rest}{b}$
    $loesung.\text{append}(a_i)$
    $a \leftarrow b$
    $b \leftarrow rest$
  return s

Mit dem erweiterten Euklidischen Algorithmus lassen sich folgende regulären Kettenbrüche finden:

- $1.857\,143 = [1;1,6,142857]$
- $1.293\,982 = [1;3,2,2,25,10,3,2,1,1,6]$
- $3.141\,593 = [3;7,16,983,4,2]$
- $1.414\,214 = [1;2,2,2,2,2,2,2,2,3,6,1,2,1,12]$

Die letzte Zahl in der Liste der Kettenbruchdarstellung ändert am wenigsten an dem Betrag der Zahl. Es liegt nun nahe, diese Zahl weg zu lassen und das Ergebnis zu untersuchen:

- $[1;1,6] = {}^{13}/{}_{7}$
- $[1;3,2,2,25,10,3,2,1,1] = {}^{98199}/{}_{75889} = 1.293982$
- $[3;7,16,983,4] = {}^{1396303}/{}_{444457} = 3.141593$
- $[1;2,2,2,2,2,2,2,2,3,6,1,2,1] = {}^{55498}/{}_{39243} = 1.414214$

Allgemein kann man von einem Kettenbruch $x = [a_0; a_1, a_2, \ldots, a_n]$ alles nach a_k weg lassen. Das nennt man dann k-te Konvergente. Betrachtet man alle Konvergenten von x, so kann man unter ihnen den Bruch finden, der mit einer festen Anzahl von Stellen im Nenner am nächsten zu x ist.

	$n=2$		$n=3$		$n=4$	
x	$\tilde{x}$	$\Delta = \|x-\tilde{x}\|$	$\tilde{x}$	$\Delta = \|x-\tilde{x}\|$	$\tilde{x}$	$\Delta = \|x-\tilde{x}\|$
1.857 143	13/7	0.000 000 14	13/7	0.000 000 14	13/7	0.000 000 14
1.293 982	22/17	0.000 135 65	559/432	0.000 000 52	11783/9106	0.000 000 01
3.141 593	311/99	0.000 178 86	355/113	0.000 000 08	355/113	0.000 000 08
1.414 214	99/70	0.000 071 71	1393/985	0.000 000 80	12397/8766	0.000 000 01

Tabelle 5: Ergebnisse der Kettenbruchsuche, mit einem maximal n-stelligem Nenner

1.5 Vereinbarkeit der Ziele

Grundsätzlich kann man auf zwei Arten versuchen, die Ziele Genauigkeit („möglichst nahe“) und Einfachheit („möglichst klein“) miteinander zu vereinbaren:

Absolut Bei einigen der beschriebenen Algorithmen haben wir bereits die Einschränkung gemacht, dass der Nenner höchstens vier Stellen haben darf, da größere Nenner in der Schule selten vorkommen. Mit der selben Begründung kann man sagen, dass Abweichungen von $\Delta < \frac{1}{9999} \approx 0.0001$ akzeptabel sind.

Relativ Alternativ kann man den relativen Fehler $\frac{|x-\tilde{x}|}{x}$ betrachten. Wenn dieser unter einer Toleranzschwelle liegt, ist die Zahl exakt genug.

Es ist in dieser Aufgabe von Vorteil, sich zuerst auf eine akzeptable Länge festzulegen und dann den besten Bruch zu finden, da es nur endlich viele Brüche mit einem höchstens n-stelligem Nenner gibt. Also gibt es unter ihnen offensichtlich eine optimale Approximation.

Es sind aber auch kompliziertere Vorhgehensweisen denkbar. So könnte der Zähler 1 oder 0 besonders gewichtet werden. Damit könnte z. B. $x = 0.005\,01$ eher zu $\tilde{x}_1 = \frac{1}{200}$ mit $|x - \tilde{x}_1| = 0.000\,01$ approximiert werden, als zu $\tilde{x}_2 = \frac{5}{998}$ mit $|x - \tilde{x}_2| = 0.000\,000\,02$ Auch den Nullen am Ende des Zählers bzw. Nenners könnte eine besondere Bedeutung zugewiesen werden.

1.6 Bewertungskriterien

- Die Kriterien für „möglichst klein“ (Einfachheit) und „möglichst nahe“ (Genauigkeit) müssen verständlich und einigermaßen präzise erklärt werden. Außerdem sollte etwas zur Vereinbarkeit der Kriterien gesagt sein.
- Das gewählte Verfahren soll gut nachvollziehbar beschrieben sein.
- Kern der Aufgabe ist, sich selbst über die Vereinbarkeit von Einfachheit und Genauigkeit Gedanken zu machen und diese (soweit geeignet) umzusetzen. Es ist deshalb nicht Sinn der Sache, eine Liste von Brüchen zu produzieren und dann dem Benutzer die Auswahl zu überlassen. Akzeptabel ist aber, wenn dem Programm eine Grenze für Genauigkeit oder Einfachheit vorgegeben wird und innerhalb dieser Grenze(n) die nach dem jeweils anderen Kriterium beste Lösung gefunden wird. Verfahren der Güteklasse von „Brute-Force II“ werden bei einem solchen Vorgehen aber tendenziell in der Nähe der gesetzten Grenzen landen; das erfüllt wiederum nicht die Erwartungen.
- Ergebnisse müssen für die in der Aufgabenstellung genannten Zahlen angegeben sein. Weitere Beispiele sind nett, aber nicht erforderlich. Die angegebenen Werte müssen natürlich die eigenen Kriterien erfüllen und sollten zusätzlich (falls die Kriterien allzu großzügig sind) bzgl. Genauigkeit und Einfachheit einigermaßen brauchbar sein.

Aufgabe 2: Tutorien

2.1 Problemstellung

Das zentrale Problem dieser Aufgabe besteht darin, zu entscheiden, ob n Tutoren, die an einigen von insgesamt n Terminen Zeit haben, so den Terminen zugeordnet werden können, dass jeder Tutor genau einen Termin betreut, an dem er selbst Zeit hat: Gesucht wird ein *Matching* zwischen Tutoren und Terminen. Eine genauere Betrachtung dieses Problems folgt im nächsten Abschnitt.

Hat man das Matching-Problem gelöst, kann darauf aufbauend die Frage bearbeitet werden, ob eine Liste von nun $n+1$ Terminen brauchbar ist. Das ist sie dann, wenn nach dem Streichen eines beliebigen Termins in der entstehenden Liste von n Terminen immer ein Matching zu finden ist. Die Brauchbarkeit kann also geprüft werden, indem für jeden der $n+1$ Termine dieser gestrichen und mit den verbleibenden n Terminen ein Matching gesucht wird.

Zu Beginn liegt aber eine Menge von $t > n$ Terminen vor. Um Teilmengen von $n+1$ Terminen auf Brauchbarkeit prüfen zu können, müssen also zuallererst alle Termin-Teilmengen dieser Größe ausgewählt werden. Dies ist ein kombinatorisches Problem, das dem „Ziehen von $n+1$ Kugeln aus einer Urne mit $n+k$ Bällen“ entspricht.

Schweißt man schließlich all diese Bestandteile zusammen, erhält man ein Programm, das die Anforderungen der Aufgabe erfüllt.

2.2 Zweidimensionales Matching

Beim zentralen Problem liegen eine Menge von n Terminen und eine von n Tutoren vor. Außerdem ist für jedes Tutor-Termin-Paar bekannt, ob der jeweilige Tutor an diesem Termin Zeit hat. Unsere Aufgabe besteht nun darin, zu entscheiden, ob jeder Tutor genau einem Termin zugewiesen werden kann, an dem er auch Zeit hat. Dieses Entscheidungsproblem wird *zweidimensionales Matching* genannt, wobei *zweidimensional* andeutet, dass wir es hier mit zwei Mengen, der Menge der Termine und der der Tutoren, zu tun haben. Ebenso denkbar ist auch ein *dreidimensionales Matching*, bei dem beispielsweise für die Tutoren zusätzlich zu den Terminen auch noch Vorlieben für Seminarräume und deren Verfügbarkeit berücksichtigt werden müssen. Im Gegensatz zu seinem großen Bruder ist der zweidimensionale Fall kein NP-vollständiges Problem und glücklicherweise in polynomieller Laufzeit, also effizienter lösbar als durch bloßes Ausprobieren aller möglichen Tutor-Termin-Zuweisungen. Tatsächlich wurde dies erst dadurch klar, dass ein solcher effizienter Algorithmus gefunden wurde. Im folgenden werden wir einen solchen Algorithmus erläutern.

Die Grundidee des Algorithmus ist es, konstruktiv nach und nach Tutor-Termin-Paare auszuwählen. Es ist klar, dass wir schließlich erfolgreich ein *Matching* gefunden haben, wenn wir n Paare ausgewählt haben und dabei jeden Termin und jeden Tutor genau einmal zuordnen. Das Auswählen neuer Paare geschieht nach bestimmten Regeln, eine davon ist es, dass niemals ein Paar gewählt wird, das einen Tutor oder einen Termin enthält, der zu einem bereits gewählten Paar gehört. Eine Auswahl von Paaren, die dieser Regel gehorcht und noch nicht n Paare enthält, ist ein sog. *partielles Matching*.

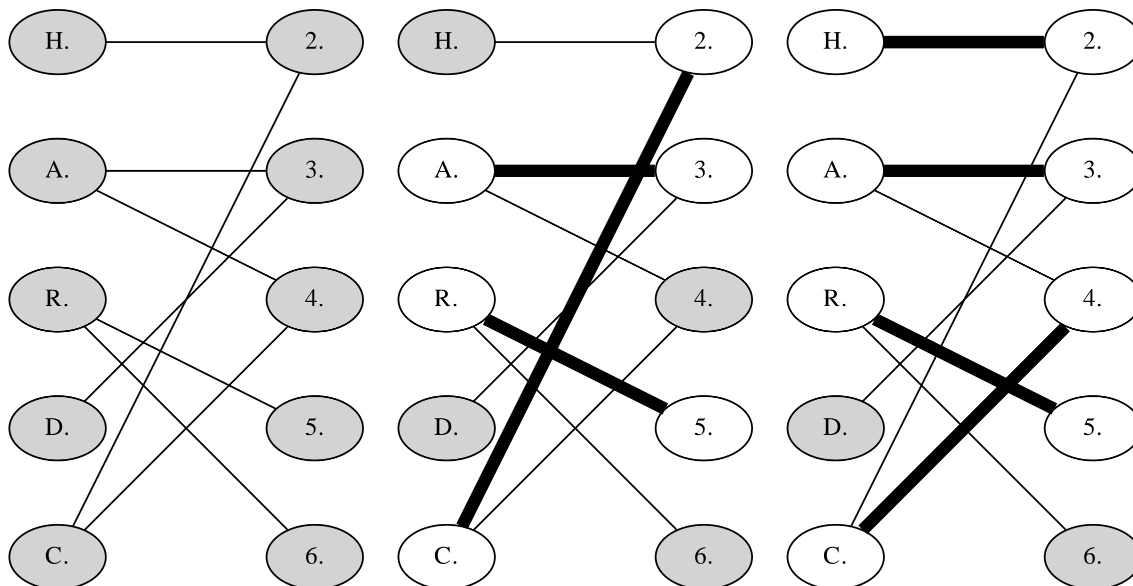


Abbildung 1: Bipartiter Graph zur Darstellung des Matching-Problems aus Tabelle 7. Isolierte Knoten sind grau gefärbt, ausgewählte Kanten sind dick dargestellt. Um vom zweiten Bild zum dritten zu gelangen, wird der *augmenting path* (dt. etwa: erweiternder Pfad) $H-2-C-4$ invertiert, danach sind keine weiteren solcher Pfade vorhanden.

Um den Ablauf des Algorithmus besser zu verstehen, bietet sich eine Darstellung der Tutor-Termin-Paare als Graph an. Dabei werden die Tutoren als Knoten auf der linken Seite und die Termine als Knoten auf der rechten Seite dargestellt (siehe Abb. 1, linke Seite). Die Kanten zwischen den Knoten repräsentieren die erlaubten Zuordnungspaare, so steht also beispielsweise Tutor H. am 2. Termin zur Verfügung, zu allen anderen Terminen aber nicht. Außerdem ist klar, dass jede Kante einen Knoten auf der linken Seite mit einem auf der rechten verbinden muss. Man spricht deshalb von einem bipartiten Graphen.

Ein im Ablauf des Algorithmus gewähltes Paar wird im Graphen als dicke Kante dargestellt. Die obige Bedingung an ein partielles Matching bedeutet dann, dass in keinem Knoten gleichzeitig zwei dicke Kanten enden.

Das Beispiel aus Tabelle 7 ohne den 1. Termin ist in der Graphdarstellung in Abbildung 1 links zu sehen.

Als letztes seien nun noch zwei Begriffe eingeführt, bevor wir uns an den eigentlichen Kern des Algorithmus heranwagen. Endet in einem Knoten kein ausgewähltes Paar, also keine dicke Kante, so sprechen wir von einem *isolierten* Knoten. Ein *Pfad* schließlich ist eine Folge von Knoten, bei der jeweils zwei aufeinanderfolgende Knoten über eine Kante verbunden sind, im obigen Bild also beispielsweise die Folge $H-2-C-4$, wohingegen die Folge $H-2-C-6$ kein Pfad ist, da zwischen C und 6 keine Kante existiert.

Die Grundlage vieler Matching-Algorithmen ist nun das sukzessive Erweitern der bereits gewählten Paare (dicke Kanten) von einer ursprünglich leeren Menge durch das Auswählen von Kanten, ohne dass die *partielle Matching*-Eigenschaft verloren geht. Dazu werden sog. *augmenting paths* gesucht, das sind Pfade im Graphen, die bei isolierten Knoten auf der linken

Seite (bei den Tutoren) beginnen und über Kanten führen, die abwechselnd bereits ausgewählt (dick) und nicht ausgewählt (dünn) sind und bei einem isolierten Knoten auf der rechten Seite enden.

Im obigen Graphen links ist also zunächst jeder Pfad, der lediglich zwei per Kante verbundene Knoten enthält, ein *augmenting path*. Der entscheidende Trick ist nun, dass durch das Invertieren aller Kanten des Pfades, also das Auswählen aller noch nicht ausgewählten Kanten und das Abwählen von schon ausgewählten Kanten, die partielle Matching-Eigenschaft gewahrt bleibt, weil jeder Knoten im Pfad, der weder Anfangs- noch Endknoten ist, an genau zwei Kanten im Pfad grenzt, von denen genau eine bereits ausgewählt ist. Nach dem Invertieren ist die jeweils andere gewählt und die vorher gewählte abgewählt. Die Endknoten sind nach Voraussetzung isoliert gewesen; durch das Invertieren grenzen auch sie schließlich an genau eine ausgewählte Kante und sind nicht mehr isoliert.

Auf diese Weise kann die Menge der gewählten Kanten stetig erweitert werden, bis schließlich keine isolierten Knoten mehr vorhanden sind. Dann existiert ein *Matching* für den Graphen, das an den gewählten Kanten abgelesen werden kann. Oder aber es gibt noch isolierte Knoten, aber keine weiteren *augmenting paths*. In diesem Fall existiert kein *Matching*.

Zur Illustration betrachten wir jetzt wieder den obigen Graphen. Durch das Hinzunehmen einiger offensichtlicher *augmenting paths*, die aus nur einer Kante bestehen, ergibt sich der mittlere Graph in Abb. 1. Ein möglicher *augmenting path* besteht dann aus den Knoten $H - 2 - C - 4$ und führt nach der Invertierung zum rechten Graphen. Auf diese Weise erhält man einen Graphen, in dem es keinen validen Pfad zwischen den letzten verbleibenden isolierten Knoten gibt. Ein Matching ist hier also nicht möglich.

Der Unterschied zwischen den meisten Algorithmen liegt darin, wie ein *augmenting path* gefunden wird. Eine naive Methode sucht dazu zunächst einen isolierten Knoten auf der linken Seite, einen mit diesem per nicht ausgewählter Kante verbundenen Knoten auf der rechten Seite und von diesem aus den Knoten auf der linken Seite, der mit dem gewählten rechten gepaart wurde, vorausgesetzt er ist nicht isoliert, in welchem Fall der Pfad gefunden wurde.

Für diesen primitiven Drei-Knoten-Pfad wird die Suche rekursiv wiederholt. Da die Rekursion nach höchstens n Stufen abbricht, ergibt sich eine Laufzeitschranke von $O(n \cdot |E|)$. Verwendet man stattdessen eine Breitensuche, um mehrere Pfade gleichzeitig zu finden, erhält man den bekannten Hopcroft–Karp-Algorithmus, der mit einer Laufzeit von $O(\sqrt{n} \cdot |E|)$ auskommt.

Mithilfe des obigen Algorithmus' kann nun der zweite Teil der Aufgabe gelöst werden, nämlich festzustellen, ob eine Tabelle mit n Tutoren und $n+1$ Terminen gültig ist. Dazu muss entweder für jeden Termin jeweils dessen Spalte aus der Tabelle herausgenommen und die übrige Tabelle auf ein Matching geprüft werden.

Alternativ kann der Matching-Algorithmus so angepasst werden, dass er auch die Eingabe von $n+1$ Terminen erlaubt und jeweils alle nahezu-vollständigen Matchings zurückgibt, die gefunden werden können. Da es in diesem Fall einen Termin "zu viel" auf der rechten Seite gibt, heißt nahezu-vollständig, dass ein Knoten isoliert bleiben darf. Für jeden Terminknoten muss es in der Liste der Matchings nun mindestens eines geben, in dem er isoliert bleibt.

	Di 8h	Di 13h	Mi 8h	Mi 13h	Do 8h
Helge	X		X		
Andrea		X	X		X
René		X		X	X
Denniz				X	

	Di 8h	Di 13h	Mi 8h	Mi 13h	Do 8h
Helge	X	X			
Andrea	X	X			
René			X	X	
Denniz				X	X

Tabelle 6: Nicht brauchbare Anmeldetabellen, da nach Entfernen des Mi 13h-Termins (links) bzw. des Di 8h-Termins (rechts) kein Matching mehr möglich ist.

2.3 Heuristiken

Nachdem wir uns der Lösung des Matching-Problems auf geteerter Straße genähert haben, sollen noch mögliche Holzwege aufgezeigt werden, um diese als mögliche Fehlerquelle zu entlarven. Dazu konzentrieren wir uns zunächst auf den zweiten Teil der Aufgabe, den „Brauchbarkeitstest“: Stelle fest, ob eine Zuordnung von n Tutoren zu $n+1$ Terminen brauchbar ist.

Das Beispiel aus der Aufgabe ist nicht brauchbar (Tab. 6, links), weil nach Entfernen des Mi 13h-Termins Denniz an keinem der übrigen Zeitpunkte zur Verfügung steht. Betrachtet man dieses Beispiel, kann leicht der falsche Eindruck entstehen, dass das Vorhandensein von mindestens zwei Terminen pro Tutor und einem Tutor pro Termin ein hinreichendes Kriterium für eine brauchbare Liste darstellt[3]. Dass dem nicht so ist, kann sich leicht am Beispiel in Tab. 6, rechts klargemacht werden: Hier hat jeder Tutor zwar zu zwei verschiedenen Terminen Zeit, ein Matching ist aber nicht möglich, wenn der erste Termin ausgelassen wird, da dann der zweite Termin entweder Helge oder Andrea zugeordnet werden muss, der jeweils andere dann aber keinen freien Termin mehr zur Verfügung hat.

Eine bessere Heuristik könnte zusätzlich verlangen, dass keine zwei Tutoren zu den gleichen Zeiten zur Verfügung stehen. Diese Aussage ist vermutlich wahr für $n = 4$,[4] ein Gegenbeispiel für $n = 5$ ist jedoch die Belegung in Tabelle 7. Wird hier der erste Termin herausgenommen, muss Denniz dem dritten Termin zugeordnet werden, sodass für Andrea nur noch der vierte bleibt, wodurch Conni den zweiten Kurs übernehmen muss, was schließlich dazu führt, dass Helge leer ausgeht.

Es ist also deutlich geworden, dass solche am Graphen abzulesenden Heuristiken nicht dazu geeignet sind, das Matching-Problem zu lösen. Vielmehr muss jede Tabelle, die durch das Weglassen eines beliebigen Termins entsteht, ein Matching haben, und ein Algorithmus wie oben vorgestellt muss benutzt werden, um dies zu entscheiden.

2.4 Die Sache mit den Urnen

Für Teil eins der Aufgabe, bei der aus der Liste von $t > n$ Terminen eine Liste von $n+1$ Terminen ausgewählt werden soll, braucht es etwas Kombinatorik und eine Methode, um alle

[3] Dass es ein notwendiges Kriterium ist, lässt sich hingegen leicht einsehen.

[4] Jedenfalls konnte ich kein Gegenbeispiel finden, sollte jemand ein solches oder einen Beweis der Vermutung finden, würde ich mich über einen Hinweis freuen! Ausprobieren aller Möglichkeiten gilt nicht.

	1.	2.	3.	4.	5.	6.
Helge	X	X				
Andrea			X	X		
René					X	X
Denniz	X		X			
Conni		X		X		

Tabelle 7: Ebenfalls nicht brauchbare Anmeldetabelle, wie das Weglassen des 1. Termins zeigt.

Möglichkeiten $m := n+1$ Kugeln aus einer Urne mit t unterschiedlichen Kugeln zu ziehen, aufzulisten. Dazu gibt es viele Standardalgorithmen, wie den effizienten von Phillip J. Chase entwickelten.[5] Aber auch naive Ansätze sind denkbar, beispielsweise folgender rekursiver:

Begonnen wird mit einem leeren (=alle Plätze sind *false*) Array von t boolschen Werten, das einer rekursiven Funktion zusammen mit den Zahlen m und t übergeben wird. Diese sucht nun nach Stellen im Array, die *false*-wertig sind, setzt sie jeweils auf *true* und ruft sich selbst auf, wobei ein Zähler erhöht wird, der als Argument mitgegeben wird. Anschließend wird der Wert wieder auf *false* gesetzt.

Wird zu Beginn der Funktion festgestellt, dass dieser Zähler den Wert m erreicht hat, wurden bereits m "Kugeln" gewählt und das übergebene Array wird als eine mögliche Auswahl gespeichert oder direkt verarbeitet.

Um bei dieser Methode auftretende doppelte Auswahlen zu vermeiden, sollten beim Suchen der *false*-Werte nur solche betrachtet werden, die sich rechts des rechtesten *true*-Werts befinden. Ein Code sagt mehr als tausend Worte; in Python-Pseudo-Code sieht das daher so aus:

```
def select_m_of_t(int m, int t, bool *a, int counter=0):
    if counter == m:
        handle(a) # callback oder Ähnliches; in a sind
                  # die gewählten Kugeln mit "true" markiert
        return

    # Finde am weitesten rechts stehenden "true"-Wert
    start_index = 0
    for i from t-1 downto 0:
        if a[i]:
            start_index = i+1
            break

    # Jede mögliche Stelle, an die nun ein Wert auf "true" gesetzt
    # werden kann, wähle aus und rufe rekursiv auf.
    for i from start_index to t:
        a[i] = True
        select_m_of_t(m, t, b, counter + 1)
        a[i] = False
```

[5] CHASE, Phillip J. "Algorithm 382: combinations of M out of N objects [G6]." Communications of the ACM 13.6 (1970): 368.

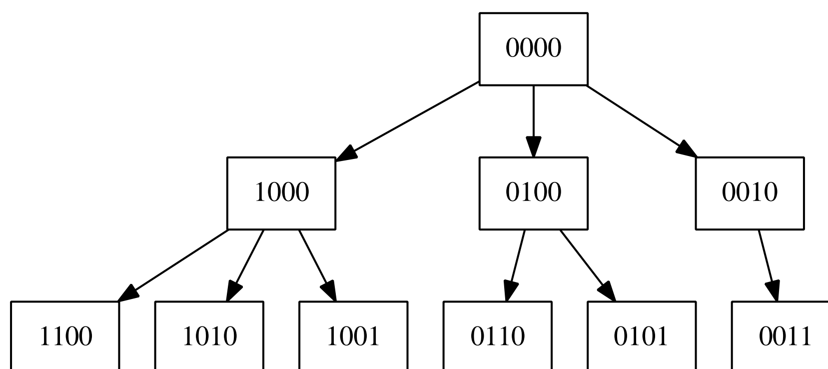


Abbildung 2: Aufrufbaum zur Erzeugung der Verteilungen von 2 Kugeln auf 4 Plätze, wobei die Beschriftungen jeweils die Belegung des Arrays a angeben. Von der Wurzel aus wird die Belegung 0001 nicht generiert, weil für diese keine valide Belegung mehr erzeugt werden kann. In der i-ten Ebene befinden sich daher $\binom{t-m+i}{i} = \binom{2+i}{i}$ Aufruf-Knoten.

Eine alternative Methode ist natürlich auch zulässig, solange sie sicherstellt, dass keine Möglichkeit der Auswahl doppelt generiert wird und nicht zu lange braucht.[6]

Die Laufzeit des rekursiven Ansatzes oben lässt sich durch Betrachtung des Aufrufbaums verdeutlichen. Dieser enthält nach Konstruktion $\binom{t}{m}$ Blätter, die jeweils in einer Tiefe von m liegen. Da nur das Abbruchkriterium einen Aufruf für m von dem für $m-1$ unterscheidet, finden sich in der Tiefe $m-1$ also $\binom{t}{m-1}$ Knoten, und so weiter, bis zur Wurzel, die alleine dasteht ($\binom{t}{0}$). So ergibt sich eine Laufzeitschranke von $O\left(\sum_{i=0}^{m}\binom{t}{i}\right)$, was immerhin besser ist als $O\left(\sum_{i=0}^{t}\binom{t}{i}\right) = O(2^t)$ im noch naiveren Fall des reinen Durchprobierens.

Aber auch obiger Algorithmus ist noch verbesserungsfähig. So sollten, falls $m > t$, stattdessen $t-m$ *false*-Werte auf ein *true*-wertiges Array verteilt werden. Eine weitere lohnenswerte Verbesserung erhält man, indem die for-Schleife statt bis t nur soweit läuft, dass die anschließend noch zu verteilenden Werte überhaupt noch Platz haben, also bis $t-(m-\text{counter})$. Mithilfe dieser Überlegung lässt sich die Laufzeit auf $O\left(\sum_{i=0}^{m}\binom{t-m+i}{i}\right) = O\left(\binom{t+1}{m}\right)$ reduzieren. Als Beispiel sei hier der Aufrufbaum für $m=2$ und $t=4$ gezeigt (Abbildung 2).

Obige Laufzeitrechnung verschweigt die Suche nach dem am weitesten rechts stehenden *true*-Wert. Diesem kann abgeholfen werden, indem diese Zahl einfach als weiterer Parameter der rekursiven Funktion übergeben wird, da die aufrufende Funktion diesen Wert kennt.

Abschließend noch ein Wort zu einer möglichen Alternativmethode: Analog zum alternativen Vorgehen zuvor, bei der alle Matchings berechnet werden, kann auch hier der Matching-Algorithmus so angepasst werden, dass er t Terminknoten zulässt und alle Matchings berechnet werden, die n Kanten enthalten, also alle Tutorknoten einem Terminknoten zuweisen. Aus der Liste dieser Matchings und den isoliert bleibenden Terminknoten kann dann ebenfalls eine Liste erzeugt werden, wozu aber wieder alle Möglichkeiten der Auswahl durchprobiert werden müssen.

[6] Das ist natürlich Ermessenssache. Definitiv *nicht* schnell genug ist das Durchprobieren aller 2^t Möglichkeiten von Belegungen eines bool-Arrays der Länge t, um abschließend nur die herauszufischen, die die korrekte Anzahl von ausgewählten Kugeln haben.

2.5 Ein- und Ausgabe

Die Eingabe kann entweder über eine grafische Benutzeroberfläche oder über eine Datei erfolgen, was sich für Konsolenprogramme anbietet. In jedem Fall sollte es aber möglich sein, den Daten (und eventuell für schöne Ausgaben auch den Tutoren) aussagekräftige Namen zu geben. Ein Eingabeformat könnte so aussehen (für das Beispiel aus der Aufgabe):

```
6        # t
4        # n
Di 8:00  # t Namen der Daten
Di 13:00
Mi 8:00
Mi 13:00
Do 8:00
Do 13:00
Helge    # n Namen der Tutoren
Andrea
René
Denniz
X-X---   # n Zeilen mit jeweils t Spalten, die die Verfügbarkeiten
   angeben
-XX-X-
-X-XX-
---X-X
```

Die Ausgabe kann dann so aussehen:

```
Di 08h, Di 13h, Mi 08h, Mi 13h, Do 08h: nicht brauchbar; Mi 13h
   nicht entfernbar!
Di 08h, Di 13h, Mi 08h, Mi 13h, Do 13h: brauchbar!
Di 08h, Di 13h, Mi 08h, Do 08h, Do 13h: nicht brauchbar; Do 13h
   nicht entfernbar!
Di 08h, Di 13h, Mi 13h, Do 08h, Do 13h: nicht brauchbar; Di 08h
   nicht entfernbar!
Di 08h, Mi 08h, Mi 13h, Do 08h, Do 13h: brauchbar!
Di 13h, Mi 08h, Mi 13h, Do 08h, Do 13h: nicht brauchbar; Mi 08h
   nicht entfernbar!
```

Das gleiche Format kann für das Programm benutzt werden, das entscheidet, ob eine Anmeldeliste brauchbar ist. Dieses erhält als Parameter eine Liste von Terminen, und das Programm gibt in geeigneter Weise seine Zustimmung oder Ablehnung bekannt. Für obiges Beispiel kann bei einem Aufruf für die Liste „Di 08h, Di 13h, Mi 08h, Mi 13h, Do 13h“ die Ausgabe folgendermaßen aussehen:

```
Di 08h, Mi 08h, Mi 13h, Do 08h, Do 13h: brauchbar!
```

2.6 Weitere Beispiele

Im Folgenden noch ein paar Beispiele, der Übersicht halber in Tabellenform.

	1.	2.	3.	4.	5.	6.
Helge	X	X				
Andrea			X	X		
René					X	X
Denniz	X		X			
Conni		X		X		

Ausgabe:

1., 2., 3., 4., 5., 6.: nicht brauchbar; 1. nicht entfernbar!

Tabelle 8: Beispiel 1

	1.	2.	3.	4.
Helge	X			X
Andrea		X		X
René			X	X

Ausgabe:

1., 2., 3., 4.: brauchbar!

Tabelle 9: Beispiel 2

	01	02	03	04	05	06	07	08	09	10
Helge				X		X		X		
Andrea			X		X		X		X	
René			X							X
Denniz		X	X							X
Conni	X									X
Oliver		X		X					X	
Rina			X		X			X		

Ausgabe (nur brauchbare Listen):

01, 02, 03, 04, 05, 06, 07, 10
01, 02, 03, 04, 05, 06, 08, 10
01, 02, 03, 04, 05, 06, 09, 10
01, 02, 03, 04, 05, 07, 08, 10
01, 02, 03, 04, 05, 08, 09, 10
01, 02, 03, 04, 06, 07, 08, 10
01, 02, 03, 04, 06, 08, 09, 10
01, 02, 03, 04, 07, 08, 09, 10
01, 02, 03, 05, 06, 08, 09, 10
01, 02, 03, 06, 07, 08, 09, 10

Tabelle 10: Beispiel 3

2.7 Bewertungskriterien

- Bei grafischen Programmen soll eine Beschreibung der Bedienoberfläche, bei nicht-grafischen Programmen muss eine genaue Beschreibung der Ein- und Ausgabeformate vorhanden sein.
- Das Verfahren bzw. die Verfahren zur Auswahl einer brauchbaren Anmeldeliste (und der Prüfung auf Brauchbarkeit) soll gut nachvollziehbar beschrieben sein.

- Insgesamt müssen die Anforderungen der Aufgabenstellung erfüllt sein. In vielen Einsendungen kommen die Dinge leider etwas durcheinander: die einen gehen direkt von $n+1$ Terminen aus (und nicht allgemeiner von $t > n$ Terminen; das Verfahren zur Auswahl von $n+1$ aus t Terminen entfällt dann), die anderen ignorieren die Bedingung, dass am Ende jeder Tutor genau einen Termin übernehmen muss.
- Zu den Anforderungen gehört auch, dass beide Aufgabenteile (Programm zum Erstellen der Listen, Programm zum Testen einer Liste) vorhanden sein müssen. Es ist aber in Ordnung, wenn das Testprogramm in Teil 1 enthalten ist und nicht mehr separat erläutert oder demonstriert wird.
- Heuristiken funktionieren nicht. Für Kriterien wie „eine Auswahlliste ist brauchbar, wenn jeder Tutor an mindestens zwei Terminen Zeit hat und es für jeden Termin mindestens einen Tutor gibt“ lassen sich schnell Gegenbeispiele finden. Leider haben nur wenige Teilnehmer die Schwächen solcher Ansätze erkannt. Falls doch, kann vom Punktabzug abgesehen werden.
- Es sollen ausreichend (mindestens drei) Beispiele dokumentiert sein, davon sollte mindestens eins komplexer als das Beispiel in der Aufgabe sein. Durch die Beispiele sollte auch klar werden, dass das Programm (bzw. das Modul) zum Testen von Anmeldelisten funktioniert.

Aufgabe 3: Vortänzer

3.1 Das Hilfsprogramm

Gesucht ist ein Programm, das bei Eingabe eines Vortänzerprogramms T und eines Imitats I die Strafpunktzahl für I ausgibt. Die Strafpunkte setzen sich wie folgt zusammen:

- Für jedes Zeichen von I gibt es drei Strafpunkte.
- Nach jeder Sekunde wird der Abstand der beiden Roboter berechnet und zur Strafpunktzahl addiert.

Der erste Anteil lässt sich leicht berechnen, indem man die Länge des Programms I mit drei multipliziert. Für den zweiten Teil müssen die beiden Programme ausgeführt und verglichen werden.

Die größte Schwierigkeit bei der Ausführung der Tanzprogramme stellen die Wiederholungsbefehle (*Schleifen*) dar. Während alle anderen Befehle linear abgearbeitet werden können, sorgen die Schleifen für Sprünge während der Ausführung. Besonders kompliziert wird es dadurch, dass die Schleifen beliebig tief geschachtelt werden können.[7] Die einfachste Methode zur Ausführung solcher Programme besteht darin, die Schleifen durch so viele Wiederholungen ihres Inneren zu ersetzen, wie die Ziffer am Anfang der Schleife angibt. Übrig bleibt ein Tanzprogramm mit den Befehlen `F`, `B`, `r`, `l` und `-`, das direkt ausgeführt werden kann.

Ersetzen der Schleifen

Erster Ansatz Man entwickelt eine Methode, um die *erste* Schleife im Programm zu finden und durch Wiederholungen ihres Inneren zu ersetzen. Diese wendet man dann solange auf das Tanzprogramm an, bis keine Schleife mehr vorhanden ist, was man an der Abwesenheit von Ziffern und Punkten erkennt.

Um den Anfang der ersten Schleife zu finden, genügt es, die erste Ziffer im Tanzprogramm zu bestimmen. Das Ende dieser Schleife muss jedoch nicht zwangsläufig der letzte Punkt im Tanzprogramm sein, wie das Gegenbeispiel `2r2Fr.r.3r.FF` zeigt. Das Ende kann man finden, indem man beginnend beim Anfang der Schleife einen Zähler auf 0 setzt und diesen beim Durchlaufen aller folgenden Zeichen für jede Ziffer um 1 erhöht und für jeden Punkt um 1 erniedrigt. Findet man beim Zählerstand von 0 einen Punkt, gehört dieser zur gesuchten Schleife.

Um nicht unnötig ineffizient zu werden, kann man das Innere von gefundenen Schleifen rekursiv von allen weiteren Schleifen bereinigen, *bevor* man es durch mehrfache Wiederholungen seiner selbst ersetzt.

[7] In der Bezeichnungsweise der theoretischen Informatik bewirken die Schleifen, dass die Sprache aller korrekten Tanzprogramme *nicht regulär* ist, also nicht mehr von endlichen Automaten erkannt werden kann, sondern Kellerautomaten benötigt werden. In dieser Hinsicht ähnelt sie der Sprache aller korrekten Klammerausdrücke (mit Wörtern wie `()(()())` oder `(()(()(())))(()()))`).

Zweiter Ansatz Während beim ersten Ansatz auch mit Laufzeitverbesserung einige Zeichen mehrfach gelesen werden, gibt es einen weiteren Ansatz, bei dem jedes Zeichen höchstens einmal gelesen wird. Dazu durchläuft man das Tanzprogramm von vorne. Sobald man auf eine Zahl trifft, wird ein rekursiver Aufruf gestartet, der das Tanzprogramm bis zum zugehörigen schließenden Punkt von Schleifen befreit. Dessen Rückgabe wird gemäß der Ziffer am Anfang vervielfacht; anschließend macht man nach der Position weiter, an der das Unterprogramm aufgehört hat, bis man einen Punkt oder das Ende des Tanzprogramms erreicht.

Ausführung der Tanzprogramme

Nun müssen die (von Schleifen bereinigten) Tanzprogramme schrittweise ausgeführt werden. Dazu speichert man am besten für jeden Roboter Position und Bewegungsrichtung und ändert diese je nach angegebenem Befehl. Nach jedem Schritt wird der Abstand der beiden Roboter voneinander bestimmt und zur Strafpunktzahl addiert. Nach Aufgabenstellung ist der Abstand „die kleinste Anzahl von Schritten (ohne Drehung), um vom einen zum anderen Punkt zu gelangen“. Bezeichnen wir die Koordinaten von Vortänzer und Imitat mit x_1 und y_1 bzw. x_2 und y_2, beträgt der Abstand also $\max(|x_2 - x_1|, |y_2 - y_1|)$.

Die Ausführung kann erst dann beendet werden, wenn *beide* Tanzprogramme fertig sind. Wurde bei einem Programm schon zuvor das Ende erreicht, bleibt der zugehörige Roboter einfach solange stehen, bis auch das andere Programm beendet wurde. Das ist wichtig, da es nach dem Beenden des ersten Programms noch weitere Strafpunkte geben kann – egal, welches der beiden Programme zuerst am Ende ankommt.

3.2 Gute Imitate

Nun gilt es, mit Hilfe des Hilfsprogramms für drei vorgegebene Tanzprogramme Imitate mit möglichst niedriger Strafpunktzahl zu finden. Das kann man von Hand machen, indem man sich z. B. die Wege der Roboter aufzeichnet. Denkbar ist aber auch eine Lösung mit dem Computer, indem man etwa alle Programme bis zu einer bestimmten Länge erzeugt und mit den Vorgabeprogrammen vergleicht. Dabei besteht eine Schwierigkeit darin, überhaupt gültige Tanzprogramme zu erhalten – entweder erzeugt man direkt nur korrekte Programme oder man baut in das Hilfsprogramm eine Fehlererkennung ein. Möchte man nur *ein* gutes Imitat erhalten, ist dieser Ansatz durchaus praktikabel, doch zur Ermittlung *aller* Imitate zum dritten vorgegebenen Tanzprogramm in akzeptabler Laufzeit ist mehr Aufwand vonnöten – das war hier aber nicht gefordert.

Erstes Tanzprogramm

Tanzprogramm: $T =$ `4FFFFrr.`
Strafpunktobergrenze: 21

Das einzige ausreichend gute Imitat ist hier $I =$ `44F.rr.` mit genau 21 Strafpunkten (wegen einer Länge von 7 Zeichen; vgl. Abbildung 3). Hierzu muss das Hilfsprogramm mit mindestens zwei geschachtelten Schleifen zurechtkommen.

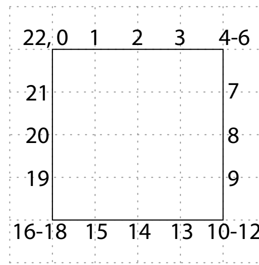


Abbildung 3: Der vom ersten Tanzprogramm und seinem Imitat zurückgelegte Weg mit den Positionen zu angegebenen Zeitpunkten (Start nach rechts).

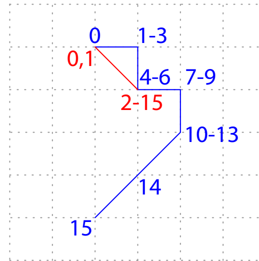


Abbildung 4: Das zweite Tanzprogramm (blau) und das Imitat mit den wenigsten Strafpunkten (rF, rot).

Zweites Tanzprogramm

Tanzprogramm: $T =$ FrrFllFrrFrrrFF
Strafpunktobergrenze: 20

Hier gibt es mehrere Lösungen, die in folgender Tabelle aufgeführt sind (vgl. Abbildung 4):

Imitat I	Strafpunkte
rF	17
F	19
FrF	20
-rF	20
r-F	20
rFl	20
rFr	20
rF-	20
FrrF	20
FllB	20

Drittes Tanzprogramm

Tanzprogramm: $T =$ lFrrFFllFFFrrFllFrrFF
Strafpunktobergrenze: 50

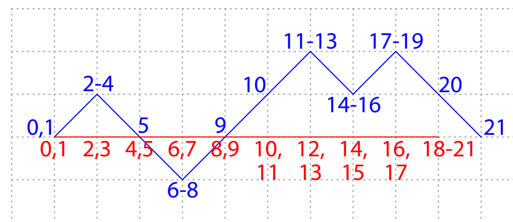


Abbildung 5: Das dritte Tanzprogramm (blau) und das Imitat mit den wenigsten Strafpunkten (`9-F.`, rot).

Durch die hohe Strafpunktobergrenze sind sehr viele Lösungen möglich: selbst bei einer Begrenzung der Imitatlänge auf acht Zeichen findet man 9706 Lösungen mit einer Strafpunktzahl von bis zu 50. Hier eine kleine Auswahl von besonders guten Imitaten, s. auch Abbildung 5:

Imitat *I*	Strafpunkte
`9-F.`	36
`8-F.`	37
`9-F.F`	38
`6-F.1F`	39
`7-F.`	39
`8-F.1F`	39
`9F-.`	39

Kleine Abwandlungen dieser Imitate haben meist immer noch Strafpunktzahlen im zulässigen Bereich.

3.3 Bewertungskriterien

Ähnlich wie bei Junioraufgabe 2 (Zollstock) gilt: Da ein Programm zum Finden von Imitaten nicht gefordert war, gibt es keine Bewertungskriterien, die zur Bestrafung des Zusatzaufwandes für ein solches Programm führen können.

- Das Verfahren zur Strafpunkteberechnung soll gut nachvollziehbar beschrieben sein.
- Das Hilfsprogramm war in der Aufgabenstellung explizit gefordert und muss beliebig geschachtelte Schleifen richtig behandeln.
- Die Strafpunktkriterien müssen korrekt umgesetzt sein, insbesondere die Formel zur Abstandsbestimmung und die Behandlung von Tanzprogrammen verschiedener Länge.
- Die Funktion des Hilfsprogramms muss an mindestens zwei eigenen Beispielen demonstriert werden; das können natürlich die für die vorgegebenen Tanzprogramme gefundenen Imitate sein. Davon sollte mindestens eines geschachtelte Schleifen oder mehrere Schleifen nebeneinander aufweisen.
- Zu jedem der drei Vorgabeprogramme muss mindestens ein Imitat mit seiner Strafpunktzahl angegeben sein. Ist die Funktionsfähigkeit des Hilfsprogramms gut nachvollziehbar, kann eine graphische Darstellung der Roboterbewegung entfallen.

Aufgabe 4: Fotoforensik

4.1 Lösungsidee

Der Verlust einer Kaffeekanne (oder auch einer Teekanne – die Kanne auf dem vorgegebenen Foto ist ja eher eine solche) ist wohl zu verschmerzen. Aber man kann sich durchaus problematischere Szenarien vorstellen, in denen man gerne feststellen möchte, ob digitale Bilder mit einer bestimmten Kamera gemacht wurden oder ob zwei oder mehrere Bilder mit der gleichen Kamera gemacht wurden. Aber digitale Bilder sind nicht mehr als Pixelmengen, abgesehen von den EXIF-Daten, die in der Regel aber nur Informationen über das Kamera-Modell enthalten. Irgendwelche versteckten Informationen über eine einzelne Kamera können darin doch nicht enthalten sein – oder?

Nun, es ist wohl nicht der Fall, dass etwa die Seriennummer einer Kamera per Steganographie in die Bilddaten hineinkodiert wird. Aber jede einzelne Kamera hinterlässt individuelle Spuren, und das war auch zu erwarten, denn ansonsten wäre die Aufgabe ja nicht lösbar gewesen.

Unter „image forensics“ werden teilweise unterschiedliche Dinge verstanden; neben der Frage nach der Zuordnung von Bildern zu Kameras kann auch die Entdeckung von Spuren digitaler Bildbearbeitung gemeint sein. Die in dieser Aufgabe gemeinte Variante von Fotoforensik ist zumindest so wichtig, dass es Unternehmen gibt, die entsprechende Software bzw. Dienstleistungen anbieten. Und natürlich gibt es auch wissenschaftliche Arbeiten zu dem Thema. Den bei Google Scholar am höchsten gelisteten Artikel zum Suchstring „digital image forensic“ haben denn auch mehrere Teilnehmer gefunden und sich davon inspirieren lassen. Auch in der Wikipedia konnte man sich beim Eintrag zu „Digitale Bildforensik“ bedienen. Kein Problem, gegen derart (zumindest ansatzweise) wissenschaftliches Arbeiten ist nichts zu sagen.

Die zentrale Idee ist, dass die Bildsensoren digitaler Kameras sozusagen einen „Fingerabdruck“ haben (von dem ja schon in der Aufgabenstellung die Rede war), den sie in jedem Bild hinterlassen. Durch leichte Varianzen in der Qualität der Pixel hinterlässt ein Sensor ein individuelles Rauschmuster in den von ihm produzierten Bildern. Das Rauschen kann man ermitteln, indem man vom ursprünglichen Bild eine „rauschfreie“ Version (in der z. B. durch Medianfilter das Rauschen entfernt wurde) abzieht. Hilfreich ist nun, wenn man das Rauschen über mehrere Bilder derselben Kamera mitteln kann. Die in den Beispieldaten gegebenen vier bis fünf Bilder pro Kamera waren für die Zwecke der Aufgabe ausreichend. Das Rauschen des fraglichen Bildes lässt sich nun mit dem gemittelten Rauschen der Probebilder der einzelnen Kameras abgleichen. Das Kannenbild korreliert diesbezüglich klar am stärksten mit den Bildern von Kamera 3: der Übeltäter ist gefunden!

Es ist aber nicht zwingend nötig, die in der Literatur beschriebenen Verfahren zu implementieren. Auch einfachere Kenngrößen sind denkbar, etwa der durchschnittliche Farbabstand benachbarter Pixel (ein auf die Sensor- bzw. Bildpixel bezogenes Vorgehen hatte die Aufgabenstellung ja recht deutlich nahegelegt). Auch hiermit lässt sich Kamera 3 als Quelle des Kannenbildes ermitteln. Wer es sich allzu einfach macht und z. B. die Farbwerte aller Pixel in einen Mittelwert verrechnet, liegt bei der Zuordnung allerdings eher falsch.

4.2 Bewertungskriterien

- Die Wahl des Verfahrens zur Bestimmung der Bildquelle sollte nachvollziehbar beschrieben und vernünftig begründet sein.
- Das Verfahren sollte nicht allzu einfach gestrickt sein, damit es eine Chance hat, die Quellkamera zu identifizieren. Wenn jemandem kein besseres Verfahren eingefallen ist, sollten die Schwächen wenigstens erkannt oder erahnt worden sein.
- Das Verfahren sollte auch mit größeren Bildmengen noch brauchbar funktionieren; bei der Verrechnung ganzer Bilder müssen in der Regel nur die passenden Pixel miteinander verrechnet werden. Nicht vollständig automatisierte Verfahren (etwa solche mit einem von Hand durchzuführenden Bildbearbeitungsschritt) fallen bei diesem Kriterium grundsätzlich durch.
- Ein selbst entwickeltes Verfahren sollte implementiert sein. Die Implementierung sollte keine Fehler enthalten, insbesondere wenn diese zu Fehlurteilen führen.
- Kamera 3 sollte eindeutig identifiziert sein oder zumindest zu den Favoriten im Kreis möglicher Quellkameras gehören.
- Die Verwendung von Verfahren aus der Literatur (inkl. Wikipedia) und von Bibliotheken zur Bildbearbeitung sind zulässig.

Aufgabe 5: Versteigerung

5.1 Das Spiel

Das Spiel *Versteigerung* besteht aus 10 Runden. In jeder Runde bieten die beiden Firmen, *Alphasoft* und *Betahard*, einen nichtnegativen Betrag an Unzen auf $2g$ Sehrteurium. Die Firma mit dem höheren Gebot erhält die $2g$ Sehrteurium, beide Firmen verlieren jedoch ihren gebotenen Betrag an Unzen. Sollten die Gebote gleich sein, wird das Sehrteurium geteilt. Zu Beginn besitzt jede Firma 1000 Unzen. Eine Firma *gewinnt* das Spiel, wenn sie nach den 10 Runden mehr Sehrteurium oder genauso viel Sehrteurium, aber mehr Unzen besitzt als die andere Firma. Sonst *verliert* diese Firma das Spiel.

Bei jedem Gebot kann eine Firma auf die eigenen Gebote sowie die Gebote des Gegners aller vorigen Runden zugreifen. Sie hat jedoch keinen Zugriff auf das Gebot des Gegners in dieser Runde. Bei dem Spiel „Versteigerung“ handelt es sich also um ein Spiel mit *perfektem Erinnerungsvermögen*. Über die Firma Alphasoft ist weiterhin bekannt, dass sie in jeder Runde 100 Unzen bietet. Diese Information macht das Spiel zu einem Spiel mit *perfekter Information*.

Ziel ist es nun, eine Künstliche Intelligenz (KI) zu entwickeln, die die Firma Alphasoft meistens besiegt. Darüber hinaus soll hier auch ein (anderer) Algorithmus für eine KI entwickelt werden, welcher die Firma Alphasoft auch dann noch schlägt, wenn im Voraus keine Information über deren Bietverhalten bekannt, das Spiel also ohne perfekte Information ist.

5.2 Lösungsideen

Die 10 Gebote der Firma Alphasoft seien im Folgenden mit $a_1, a_2, ..., a_{10}$ bezeichnet. Die eigenen Gebote seien mit $k_1, k_2, ..., k_{10}$ bezeichnet.

Konstant-bietende Algorithmen

Für den Fall $a_1 = a_2 = ... = a_{10} = 100$ gehört der Algorithmus, der $k_1 = k_2 = ... = k_9 = 101$ und $k_{10} = 91$ bietet, wohl zu den trivialsten.

Es ist (sehr) leicht zu erkennen, dass es zum Besiegen von Alphasoft genügt, in mindestens sechs Runden mehr als 100 Unzen zu bieten. Obiger Algorithmus erfüllt dieses hinreichende Kriterium für einen Sieg ebenso wie alle anderen Algorithmen, die in sechs Runden einen konstanten Betrag zwischen 101 und 166 Unzen (also beispielsweise 6 mal 142 Unzen) bieten. (In den anderen Runden bieten diese entweder den konstanten Betrag, oder den verbleibenden Betrag an Unzen, sofern dieser geringer ist, als der eigentlich zu bietende Betrag.) Auch Alphasoft verfährt nach einem konstant-bietenden Algorithmus.

Im allgemeineren Fall, dass die Gebote von Alphasoft nicht bekannt sind, sind konstant-bietende Algorithmen offensichtlich wenig geeignet.

Linear-bietende Algorithmen

Denkbar sind auch Algorithmen, die nach dem Muster $k_n = a \pm (n-1) \times c \ a,c \in \mathbb{N}$ bieten. Mit geeigneten a und c kann dieser Algorithmus in der Lage sein, bestimmte konstant-bietende Algorithmen, insbesondere den von Alphasoft, zu besiegen.

Zum Beispiel soll der Algorithmus $(a^1{}_n)$, der in 6 Runden 166 Unzen bietet, geschlagen werden. Der zu konstruierende Algorithmus muss offensichtlich in nur zwei Runden mehr als 166 Unzen bieten, um diese zu gewinnen; in vier anderen Runden genügt es, mehr als $1000 - 6 \times 166 = 4$ Unzen zu bieten.

Der folgende linear-bietende Algorithmus genügt dieser Bedingung: $k_n = 7 + (n-1) \times 20$. Dieser Algorithmus bietet also erst 7 und dann 27, 47, 67, 87, 107, 127, 147, 167 und 187 Unzen. Am Ende behält er sogar noch 30 Unzen übrig. Weiterhin gewinnt dieser Algorithmus die letzten beiden Runden gegen $(a^1{}_n)$. Außerdem gewinnt dieser Algorithmus die 4 Runden, in denen $(a^1{}_n)$ weniger als 5 Unzen bietet. Insgesamt schlägt dieser Algorithmus also $(a^1{}_n)$, sofern dieser in den letzten beiden Runden 166 Unzen bietet. (Anstatt $k_n = 7 + (n-1) \times 20$ kann auch $k_n = 187 - (n-1) \times 20$ verwendet werden, um eine umgekehrte Bietfolge zu erhalten. In diesem Fall müssen die ersten beiden Gebote des Gegners 166 sein, damit der Algorithmus gewinnt.)

Dieser Algorithmus besiegt Alphasoft: Alphasoft gewinnt und verliert jeweils fünf Runden, hat am Ende jedoch weniger Unzen übrig und verliert somit.

„Immer-eins-mehr"-Algorithmen

Eine weitere Klasse von Algorithmen bietet immer einen bestimmten Betrag mehr als das letzte Gebot des Gegners, also $k_n = d + a_{n-1}$, mit $a_0 = 0$. Diese Algorithmen sind konstant-bietenden Algorithmen offensichtlich überlegen. Ebenso sind sie linear-bietenden Algorithmen (l_n), $l_n = a + (n-1) \times c$ klar unterlegen, wenn $c < d$ oder wenn $l_n = a - (n-1) \times c$. Ansonsten sind die linear-bietenden Algorithmen unterlegen.

Adaptive Algorithmen

Kennt man obige Arten von Algorithmen, so ist es möglich, aus den ersten 2 bis 4 Runden Vorhersagen für die restlichen Runden zu treffen. Man speichert sich also zunächst die ersten 2 Gebote des Gegners und versucht, in diesen ein Muster zu erkennen, das auf die Bietfolge eines der obigen Algorithmenklassen „passt". Ist dies nach 2 Runden noch nicht eindeutig möglich, so schaut man sich das 3. Gebot des Gegners an, usw. Spätestens ab der 5. Runde sollte der Algorithmus jedoch anfangen, Gebote abzugeben, von denen er erwartet, dass sie über denen des Gegners liegen. Weiterhin kann und sollte der Algorithmus die Gebote des Gegners mit einbeziehen, um mögliche Fehler zu korrigieren.

Insgesamt können Algorithmen dieser Klasse theoretisch alle einfacheren, deterministischen Algorithmen schlagen.

Randomisierte Algorithmen

Einen konstant-bietenden Algorithmus mit einem zufällig-bietenden Algorithmus sicher zu besiegen, scheint auf den ersten Blick vielleicht etwas abwegig. Von einem Algorithmus, der in jeder Runde einen zufälligen Betrag $k \geq 0$ bietet, ist dies sehr wohl nicht garantiert, wohl aber von dem Algorithmus, der in jeder Runde einen Betrag $100 < k < 200$ bietet, um nur ein Beispiel zu nennen. Auch ist es möglich, den oben bereits genannten Algorithmen eine zufällige Komponente zu geben. So kann beispielsweise bei einem linear-bietenden Algorithmus der Betrag c (also der Betrag, um den sich die Gebote in jeder Runde erhöhen) zufällig sein.

Es ist wohl einsichtig, dass insbesondere adaptive Algorithmen den randomisierten Algorithmen unterlegen sind. Auch kann durch geschickte Wahl der Parameter der Zufallsgröße erreicht werden, dass die randomisierten Algorithmen ihren deterministischen Entsprechungen überlegen sind.

„Die Alternative“-Algorithmen

Kennt man obige Algorithmen, so besteht zum einen die Möglichkeit, einen adaptiven Algorithmus zu entwickeln. Alternativ[8] kann man auch versuchen, sich eine Bitfolge zu konstruieren, von der man sicher sein kann, dass sie die meisten der Vertreter obiger Algorithmen schlagen kann. Aus eben den obigen Algorithmen ergeben sich folgende Anforderungen an diese Bietfolge:

- Mindestens 6 mal soll ein Betrag $k > 100$ geboten werden, um so zu gewährleisten, dass Alphasoft besiegt wird.
- Nach den 10 Geboten sollen Unzen übrig bleiben, um so bei Gleichstand ggf. doch noch zu gewinnen.
- Auf hohe Gebote sollen niedrige Gebote folgen und umgekehrt, um so „Immer-eins-mehr“-Algorithmen zu besiegen. Durch geschickte Wahl der gebotenen Beträge besiegt man auf diese Weise auch linear-bietende Algorithmen.
- Kein Gebot sollte 0 sein, insbesondere nicht die letzten, da manche Algorithmen dann eventuell schon keine Unzen mehr übrig haben.
- Adaptive Algorithmen verlieren sowieso, benötigen also keine weitere Fürsorge. (Für adaptive Algorithmen verhält sich ein „Die Alternative“-Algorithmus wie ein randomisierter, mit den für den adaptiven Algorithmus daraus resultierenden Problemen.)

Sicher kann man noch weitere Kriterien aufstellen, um noch mehr Algorithmenklassen in ihren verschiedenen Ausführungen zu besiegen. Obige Kriterien genügen aber, um die meisten Vertreter der genannten Algorithmenklassen zu besiegen.

Eine nach den obigen Regeln entworfene Bietfolge kann nun beispielsweise folgendermaßen Aussehen: $10, 165, 101, 2, 166, 4, 14, 170, 142, 165$, wobei 61 Unzen übrig bleiben.

[8] Daher der zugegeben merkwürdige Name dieser Algorithmenklasse.

Abschließende Bemerkungen

Die hier vorgestellten Verfahren sind allesamt als möglich, jedoch nicht als einzig möglich anzusehen. Andere Verfahren sind also denkbar, insbesondere bei den randomisierten Algorithmen bestehen viele Variationsmöglichkeiten.

So stark oder schwach ein Algorithmus auf dem Papier erscheinen mag, die wirkliche Güte eines Algorithmus lässt sich wohl erst im Turnier erkennen. Dass ein Algorithmus möglichst viele andere Algorithmen schlägt, stand bei dieser Aufgabe aber nicht im Vordergrund.

5.3 Bewertungskriterien

Diese Aufgabe mag auf den ersten Blick trivial erscheinen. Umso wichtiger ist daher eine gute Dokumentation. Im einzelnen sollte Folgendes beachtet werden:

- Das realisierte Verfahren muss Alphasoft garantiert besiegen – und es sollte erwähnt werden, warum dies gelingt.
- Die Überlegungen zur Wahl der Strategie(n) dürfen nicht nur an Alphasoft ausgerichtet sein, sondern müssen auch andere potenzielle Gegner berücksichtigen. Ein siegreiches Abschneiden gegen Alphasoft genügte zwar für das „erfolgreiche Lösen“ der Aufgabe (für das es ohne weitere Mängel vier von fünf möglichen Punkten gibt, die im Durchschnitt für die Qualifikation zur 2. Runde genügen). Der geforderte Einsatz der Lösung im Turniersystem macht aber ein weitergehendes Nachdenken zwingend erforderlich.
- Insgesamt sollten die Wahl der Strategie(n) begründet und die Strategie(n) nachvollziehbar beschrieben sein.
- Es sollte deshalb anhand von ausreichend vielen, aussagekräftigen Beispielen gezeigt werden, welche anderen Algorithmen der Algorithmus schlägt, und insbesondere welche nicht.
- Da die Aufgabenstellung den Einsatz der Lösung im Turniersystem fordert, ist es offensichtlich, dass die im Turniersystem abgegebene KI in der Einsendung benannt werden sollte. Das ist aber in so wenigen Einsendungen der Fall, dass das Fehlen des KI-Namens nur als leichter Mangel angemerkt ist.
- Der Algorithmus sollte in Code umgesetzt werden, der im Turniersystem lauffähig ist (wohingegen ein Beteiligung am Turnier nicht gefordert ist). Dies beinhaltet auch, dass die KI in der gegebenen Berechnungszeit zu einer Lösung kommt, keine unerwarteten Ausnahmen wirft o.Ä.

Aus den Einsendungen: Perlen der Informatik

Allgemeines

Worte des Wettbewerbs: Alphabeet, Tolleranz

Die Lösungsidee wurde sowohl nicht in Python als auch nicht in C# umgesetzt.

Ursprünglich war mir eine Schleife lieb ...

Jedenfalls habe ich es mit nicht allzu magischen Mitteln geschafft, den gemeinen Bug zu beheben.

Dann kommt eine if-Schleife.

Sie halten hier eine von vielen Lösungen des Bundeswettbewerbs Informatik in der Hand. Herzlichen Glückwunsch! Sie müssen wissen, dass mich diese Seiten Unmengen an Schlaf, Extraarbeit und Zeit, welche für die Schule gefehlt hat, kosteten, und ich bin stolz, Ihnen diese Seiten nun vorstellen zu dürfen. ... bevor ich Ihnen das Produkt eines rauchenden Kopfes zur Wertung vorführen darf:

Songwriter

Codeschnipsel: `konso = "aeiou"`

Ich übernehme keine Haftung für langweilige, sinnlose oder bescheuerte Texte.

Die Lösungsidee wurde in einem Programm in der Programmiersprache Java umgesetzt. Die Lösungsidee wurde Bottom-Up implementiert. Im Quelltext stehen die Methoden in umgekehrter Reihenfolge, also Top-Down.

Natürlich hätte man alles Weitere auch innerhalb des Konstruktors machen können, jedoch ist die Auslagerung dessen weitaus konstruktiver ...

Gesammeltes „pseudocooles Zeugs“: bagohart; you don’t simply walk into mordor; uiaeeeee-eeeee!; Das Ghetto ist hart, man!; Ich bin adoptiert; bwinf

Zollstock

Codeschnipsel: `//Ausgabe der Lösung (eigentlich unwichtig)`

Da Raluca offenbar in einem handwerklichen Betrieb (deswegen der Zollstock) eine Ausbildung macht, hat sie kein großes Verständnis für Informatik. Ihr Verlobter ist bestimmt Mathematiker oder Informatiker und hat, bevor er seine Aussage traf, alles zu Hause durchgerechnet. *Ich habe irgendwann aufgehört, die in diesen zwei Sätzen enthaltenen Klischees zu zählen.*

Rationalisierung

Henry Ford würde sich freuen. *Anmerkung zum Aufgabentitel*

Damit die Endlosschleife auch ein Ende hat, besitzt die Prozedur 3 Abbruchkriterien.

Die zu brechende Zahl ist 3,1415... = Pi

Um den abstrakten Datentyp „Bruch“ zu speichern wird – wie letztes Jahr in Saarbrücken bei Prof. H. gelernt – eine Klasse erstellt. *Die Teilnehmercamps haben also doch einen gewissen Nutzen!*

... sollte man durch das Kürzen durch den „kleinsten-gemeinsamen-Teiler“(ggT) erreichen können

„möglichst nahe“ bedeutet **gleich**

... kann man auf zwei Arten versuchen, die Ziele Genauigkeit und Exaktheit miteinander zu vereinbaren. *Aus der ersten Fassung dieser Lösungshinweise – seufz!*

Tutorien

Tutoren seien im Folgenden Objekte der Begierde. *Das hat eine Teilnehmer*in *geschrieben!*

Das Eingabeformat des Programms ist so gestaltet, dass, so lange das Eingabeformat nicht zwischendurch geändert wird, alle Formen möglich sind.

Man unterteilt die vier Tutoren ...

Vortänzer

Vertänzer

Codeschnipsel:

```
Label1.Caption := 'Das war Quatsch' //"humorvolle" Fehlermeldung
```

Anschließend werden sowohl das originale Tanzprogramm als auch das Imitat ausgerollt.

Wenn der Roboter eine Anweisung ausführen will, ruft er `doMove` auf und übergibt sich ...

Da es sich um ein einfaches Simulationsprogramm handelt, verzichte ich darauf, einen nicht vorhandenen / offensichtlichen Algorithmus zu erläutern.

Fotoforensik

Das Programm ist leider in `/dev/null` verschwunden, nachdem ich gemerkt habe, dass es mich der Lösung nicht näher brachte.

Versteigerung

... die zehn Gebote ... *Klar, das musste kommen!*

Adaptive Algorithmen verlieren sowieso, brauchen also keine weitere Fürsorge. *Wie hartherzig!*

Da der Gegner kein Mensch ist, kann man davon ausgehen, dass er nicht einfach „nach Gefühl“ bietet. *Und was ist mit der ProgrammiererIn?*

Das Chaos besiegt jeden Computer.