**39. Bundeswettbewerb Informatik 2020/2021, 1. Runde**

# Lösungshinweise und Bewertungskriterien

Bundeswettbewerb Informatik

## Allgemeines

Es war sehr erfreulich, dass sich wieder besonders viele die Zeit zur Bearbeitung der Aufgaben genommen und am Bundeswettbewerb Informatik teilgenommen haben. Den Veranstaltern ist bewusst, dass in der Regel viel Arbeit hinter der Erstellung einer Einsendung steckt.

Die BewerterInnen würdigten die Leistungen der TeilnehmerInnen so gut wie möglich. Dies wurde ihnen nicht immer leicht gemacht, insbesondere wenn die Dokumentation nicht die im Aufgabenblatt genannten Anforderungen erfüllte. Bevor mögliche Lösungsideen zu den Aufgaben und Einzelheiten zur Bewertung beschrieben werden, soll deshalb im folgenden Abschnitt auf die Dokumentation näher eingegangen werden; vielleicht helfen diese Anmerkungen für die nächste Teilnahme.

Wie auch immer die Einsendung bewertet wurde, es sollte nicht entmutigen! Allein durch die Arbeit an den Aufgaben und ihren Lösungen hat jede Teilnehmerin und jeder Teilnehmer einiges gelernt; diesen Effekt sollte man nicht unterschätzen. Selbst wenn man zum Beispiel aus Zeitmangel nur die Lösung zu einer Aufgabe einreichte, so erhält man eine Bewertung der Einsendung, die für die Zukunft hilfreich sein kann.

Die Bearbeitungszeit für die 1. Runde beträgt etwa drei Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der Dokumentation der Aufgabenlösungen zu vermeiden. Einige Aufgaben sind oft schwerer zu bearbeiten, als sie auf den ersten Blick erscheinen mögen. Erst in der konkreten Umsetzung einer Lösungsidee und Testen von Beispielen stößt man manchmal auf weitere Besonderheiten bzw. noch zu lösende Schwierigkeiten, was dann zusätzlicher Zeit bedarf.

Noch etwas Organisatorisches: Sollte der Name auf Urkunde oder Teilnahmebescheinigung falsch geschrieben sein, ist er auch im AMS (login.bwinf.de) falsch eingetragen. Die Teilnahmeunterlagen können gerne neu angefordert werden; dann aber bitte vorher den Eintrag im AMS korrigieren.

### Dokumentation

Die Zeit für die Bewertung ist leider begrenzt, weshalb es unmöglich ist, alle eingesandten Programme gründlich zu testen. Die Grundlage der Bewertung ist deshalb die Dokumentation, die, wie im Aufgabenblatt beschrieben, für jede bearbeitete Aufgabe aus den Teilen *Lösungsidee*, *Umsetzung*, *Beispielen* und *Quellcode* bestehen soll. Leider sind die Dokumentationen bei vielen Einsendungen sehr knapp ausgefallen, was oft zu Punktabzügen führte, die das Erreichen der 2. Runde verhinderten. Grundsätzlich kann die Dokumentation einer Aufgabe als „sehr unverständlich oder nicht vollständig“ bewertet werden, wenn die meisten Inhalte kaum verständlich sind oder Teile wie Lösungsidee, Umsetzung oder Quellcode komplett fehlen.

Die Beschreibung der *Lösungsidee* sollte keine Bedienungsanleitung des Programms oder eine Wiederholung der Aufgabenstellung sein. Stattdessen soll erläutert werden, welches fachliche Problem hinter der Aufgabe steckt und wie dieses Problem grundsätzlich mit einem Algorithmus sowie Datenstrukturen angegangen wurde. Ein einfacher Grundsatz ist, dass Bezeichner von Programmelementen wie Variablen, Methoden etc. nicht in der Beschreibung einer Lösungsidee verwendet werden, da sie unabhängig von solchen technischen Realisierungsdetails ist. Wenn die Beschreibungen in der Dokumentation nicht auf die Lösungsidee eingehen oder bzgl. der Lösungsidee kaum nachvollziehbar sind, kann es zu einem Punktabzug führen, weil das „Verfahren unzureichend begründet bzw. schlecht nachvollziehbar“ ist.

Ganz besonders wichtig sind die vorgegebenen (und ggf. weitere) *Beispiele*. Wenn Beispiele und die zugehörigen Ergebnisse in der Dokumentation fehlen, führt das zu Punktabzug. Zur Bewertung ist für jede Aufgabe vorgegeben, zu wie vielen (und teils auch zu welchen) Beispielen korrekte Programmausgaben/-ergebnisse in der Dokumentation erwartet werden. Die Ergebnisse, die für die vorgegebenen Beispiele in der Dokumentation angegeben werden, sollten alle korrekt sein. Punktabzug für falsche Ergebnisse gibt es aber nur, wenn für den ursächlichen Mangel kein Punkt abgezogen wurde.

Es ist nicht ausreichend, Beispiele nur in gesonderten Dateien abzugeben, ins Programm einzubauen oder den Bewertern sogar das Erfinden und Testen von geeigneten Beispielen zu überlassen. Beispiele sollen die Korrektheit der Lösung belegen, und es sollen damit auch Sonderfälle gezeigt werden, die das Programm entsprechend behandeln kann.

Auch *Quellcode*, zumindest die für die Lösung wichtigen Teile des Quellcodes, gehört in die Dokumentation; Quellcode soll also nicht nur elektronisch als Code-Dateien (als Teil der Implementierung) der Einsendung beigefügt werden. Schließlich gehören zu einer Einsendung als Teil der Implementierung *Programme*, die durch die genaue Angabe der Programmiersprache und des verwendeten Interpreters bzw. Compilers möglichst problemlos lauffähig sind und hierfür bereits fertig (interpretierbar/kompiliert) vorliegen, mit allen notwendigen Eingabedaten/-dateien (z. B. für die Beispiele). Die Programme sollten vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner hinsichtlich ihrer Lauffähigkeit getestet werden.

Hilfreich ist oft, wenn die Erstellung der Dokumentation die Programmierarbeit begleitet oder ihr teilweise sogar vorangeht. Wer es nicht ausreichend schafft, seine Lösungsidee für Dritte verständlich zu formulieren, dem gelingt meist auch keine fehlerlose Implementierung, egal in welcher Programmiersprache. Daher kann es nützlich sein, von Bekannten die Dokumentation auf Verständlichkeit hin lesen zu lassen, selbst wenn sie nicht vom Fach sind.

Wer sich für die 2. Runde qualifiziert hat, beachte bitte, dass dort deutlich kritischer als in der 1. Runde bewertet wird und höhere Anforderungen an die Qualität der Dokumentation und Programme gestellt werden. So werden in der 2. Runde unter anderem eine klar beschriebene Lösungsidee (mit geeigneten Laufzeitüberlegungen und nachvollziehbaren Begründungen), übersichtlicher Programmcode (mit verständlicher Kommentierung), genügend aussagekräftige Beispiele (zum Testen des Programms) und ggf. spannende eigene Erweiterungen der Aufgabenstellung (für zusätzliche Punkte) erwartet.

## Bewertung

Bei den im Folgenden beschriebenen Lösungsideen handelt es sich um Vorschläge, aber sicher nicht um die einzigen Lösungswege, die korrekt sind. Alle Ansätze, die die gestellte Aufgabe

vernünftig lösen und entsprechend dokumentiert sind, werden in der Regel akzeptiert. Einige Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall erwartet werden. Zu jeder Aufgabe werden deshalb im jeweils letzten Abschnitt die Kriterien näher erläutert, auf die bei der Bewertung dieser Aufgabe besonders geachtet wurde. Die Kriterien sind in der Bewertung, die man im AMS einsehen kann, aufgelistet und geben an, inwieweit die Bearbeitung einer Aufgabe die einzelnen Bewertungskriterien erfüllt. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation, die oben erklärt sind.

In der 1. Runde geht die Bewertung von fünf Punkten pro Aufgabe aus, von denen bei Mängeln Punkte abgezogen werden. Da es nur Punktabzüge gibt, sind die Bewertungskriterien meist negativ formuliert. Wenn das (Negativ-)Kriterium erfüllt ist, gibt es einen Punkt oder gelegentlich auch zwei Punkte Abzug; ansonsten ist die Bearbeitung in Bezug auf dieses Kriterium in Ordnung. Wurde die Aufgabe nur unzureichend bearbeitet, wird ein gesondertes Kriterium angewandt, bei dem es bis zu fünf Punkten Abzug gibt. Im schlechtesten Fall wird eine Aufgabenbearbeitung mit 0 Punkten bewertet.

Für die Gesamtpunktzahl sind die drei am besten bewerteten Aufgaben 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 einen 3. Preis mit 9 bis 11 Punkten. Mit einem 1. oder 2. Preis ist man für die 2. Runde qualifiziert. Kritische Fälle mit nur 11 Punkten sind bereits sehr gründlich und mit viel Wohlwollen geprüft. Leider ließ sich aber nicht verhindern, dass Teilnehmende nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden; dies ist ziemlich riskant, da sich leicht Fehler einschleichen.

Auch wurde in einigen Fällen die Regelung zur Bearbeitung von Junioraufgaben als Teil einer Einsendung zum Bundeswettbewerb Informatik nicht beachtet. Hierzu ein Zitat aus dem Mantelbogen des Aufgabenblatts: „Die etwas leichteren Junioraufgaben dürfen nur von SchülerInnen vor der Qualifikationsphase des Abiturs bearbeitet werden.“ Nur unter Einhaltung dieser Bedingung können Bearbeitungen von Junioraufgaben im Bundeswettbewerb gewertet werden.

## Danksagung

Alle Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt: Peter Rossmanith, Hanno Baehr, Jens Gallenbacher, Rainer Gemulla, Torben Hagerup, Christof Hanke, Thomas Kesselheim, Arno Pasternak, Holger Schlingloff, Melanie Schmidt sowie (als Gast) Wolfgang Pohl.

An der Erstellung der im folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Fabian Aiteanu (Junioraufgabe 1), Marian Dietz und Simon Schwarz (Junioraufgabe 2), Jonathan Baumann (Aufgabe 1), Nils Langius (Aufgabe 2), Manuel Gundlach (Aufgabe 3), Hans-Martin Bartram (Aufgaben 4 und 5) und Alexandru Duca (Aufgabe 5). Allen Beteiligten sei für ihre Mitarbeit hiermit ganz herzlich gedankt.

# Junioraufgabe 1: Passwörter

## J1.1 Lösungsidee

Die Aufgabenstellung verlangt drei Regeln, nach denen gut aussprechbare und merkbare Passwörter erzeugt werden sollen. Diese Forderung wird hier vorsichtig umgangen; es wird vor allem ein Konstruktionssystem entwickelt, dass zur gewünschten Art von Passwörtern führt.

Die wichtigste Regel für ein gut auszusprechendes Passwort besagt, dass das Wort dem Klang der deutschen Sprache entsprechen soll. Diese Regel ist sehr abstrakt und erfordert eine detaillierte Ausarbeitung, die weitere Regeln eher implizit statt explizit mit einschließt. Bei Betrachtung eines längeren Wortes wie „Som·mer·fe·ri·en“ kann man leicht erkennen, dass ein Wort aus Silben besteht, die jeweils ähnlich aufgebaut sind und als eine Einheit ausgesprochen werden. Silben sind jeweils kurz und haben in diesem Beispiel eine der drei Grundstrukturen

- CV: Konsonant + Vokal
  - fe, ri
- VC: Vokal + Konsonant
  - en
- CVC: Konsonant + Vokal + Konsonant
  - Som, mer

Diese Silbenstrukturen kann man im weiteren Sinne als „Regeln“ im Sinne der Aufgabenstellung verstehen. Es gibt durchaus weitere Strukturen und Sonderfälle, die in der Umsetzung erläutert werden. Insbesondere wird zusätzlich eine „Silbenstruktur“ N verwendet, die für eine ein- bis zweistellige Zahl steht.

Die Idee besteht nun darin, zuerst eine zufällige Anzahl von Silben zu wählen. Für jede der Silben wird eine zufällige Struktur gewählt. Anschließend werden die Silbenstrukturen nacheinander durch zufällige Werte aus ihrem jeweiligen Bereich ersetzt, also wird z. B. für ein „Konsonant“ ein „n“ eingesetzt. Abschließend wird das generierte Passwort ausgegeben.

Ein Beispiel: Das Passwort soll aus drei Silben bestehen, als Strukturen werden CVC, N und CV gewählt. Daraus entsteht ganz zufällig das Passwort `hal42da`.

Doch kennt die deutsche Sprache nicht auch komplexere Silbenstrukturen wie CVCC (Kopf, Dach), CCVC (Stab, Blut) oder gar CCCCVCC (Schnitt)? Die Frage ist nicht leicht zu beantworten, schließlich beschäftigen sich ganze Fachgebiete (Phonologie, Phonotaktik) damit. Um es kurz zusammenzufassen: Man kann selbst für einfache Fälle wie CCV nicht beliebige Buchstaben einsetzten. Wenn man als ersten Buchstaben ein *m* wählt, so kann im Deutschen kein zweiter Konsonant darauf folgen. Es gibt unzählige Regeln dafür, welche Laute innerhalb einer Silbe wie angeordnet werden können, und sie benötigen letztlich ein komplettes Wörterbuch mit erlaubten Kombinationen. Wir wollen uns daher auf die oben genannten Silbenstrukturen beschränken. Diese sind auf jeden Fall mit allen einsetzbaren Werten aussprechbar. Aber solche Wörterbuch-Methoden sind als Lösung für diese Aufgabe natürlich genausogut denkbar!

Auf das Generieren von Großbuchstaben und Sonderzeichen wird verzichtet, da sie für den Benutzer außerhalb von bekannten Begriffen eher schwierig zu merken sind.

## J1.2 Umsetzung

Das zu erstellende Programm benötigt keine Benutzerinteraktion, außer zum Starten. Zuerst wird eine zufällige Anzahl an Silben im Bereich 5-7 ermittelt (**GenerateRandomValues**). Diese Grenzen wurden gewählt, da ein eigener Test gezeigt hat, dass zufällige Wörter ab 8 Silben und damit etwa 20 Buchstaben doch sehr schwer zu merken sind. Bei weniger als 4 Silben ist dass Passwort jedoch sehr kurz und daher weniger sicher.

Für jede der Silben wird anschließend eine Struktur aus den vier verfügbaren Strukturen (CV, VC, CVC, N) zufällig ermittelt (**GenerateSyllableStructure**). Hierbei steht **C** für engl. *consonant*, **V** für *vowel* und **N** für *number*.

Bis zu dieser Stelle hat der Algorithmus eine Silbenstruktur ermittelt, beispielsweise CVC-CV-CV-N-CVC. Falls benachbarte N auftreten, wird das zweite N durch CV ersetzt, damit keine Zahlenkolonnen entstehen können. Anschließend werden die Elemente der Struktur durch Phoneme aus der jeweiligen Klasse ersetzt (**GenerateLetters**). *Phoneme* sind einzelne hörbare Laute, die in der Schrift auch durch mehr als ein Zeichen dargestellt werden können. So gehören zu den Vokal-Phonemen neben a, e, i, o, u auch ie, ei, ey, au und eu. Bei den Konsonant-Phonemen gehören u. a. auch *ch* und *sch* dazu. Das *c* wurde absichtlich weggelassen, das es im Deutschen nicht alleine stehen kann, sondern nur in den sogenannten Clustern *ch* und *ck*, die wiederum teilweise nicht am Anfang einer Silbe stehen können. Ein N-Strukturelement wird durch eine Zahl im Bereich von 1 bis 99 ersetzt.

Nachdem alle Elemente ersetzt wurden, wird das Ergebnis dem Benutzer angezeigt.

### Speicherverbrauch und Laufzeit

Der vorgestellte Algorithmus hat keine expliziten Parameter. Implizit ist lediglich die Anzahl an zu verwendenden Silben für das Passwort vorgegeben, die beispielhaft im Bereich 5-7 gewählt wurde. Speicherplatz und Laufzeit hängen jeweils linear davon ab, also $O(n)$.

## J1.3 Shannon-Entropie und die Sicherheit von Passwörtern

Dieser Abschnitt beschäftigt sich etwas tiefer mit der Sicherheit und dem Maß für Sicherheit von Passwörtern. Eine solche theoretische Analyse war für die Bearbeitung der Aufgabe nicht gefordert, aber wir wollen sie interessierten Lesern nicht vorenthalten.

Schauen wir uns zunächst einmal das Passwort vom Anfang der Aufgabenstellung an. Es besteht aus 11 Zeichen, enthält Groß- und Kleinbuchstaben, Ziffern, und Sonderzeichen:

`Dfq4T&o%%kM`

Wie sicher ist dieses Passwort? Wenn wir davon ausgehen, dass dieses Passwort generiert wurde, indem 11 mal gleichverteilt zufällig aus der Menge $\{\mathrm{A},\ldots,\mathrm{Z},\mathrm{a},\ldots,\mathrm{z},0,\ldots,9,\&,\%,\ldots\}$, also einer Menge mit 72 Zeichen (vereinfachend 10 Sonderzeichen angenommen) gewählt wurde, dann ist dieses Passwort eines aus

$$72^{11} \approx 2{,}7 \cdot 10^{20}$$

möglichen und gleichverteilten Passwörtern.

Die Anzahl an möglichen (gleichverteilten!) Passwörtern ist ein gutes anschauliches Maß für die Stärke eines Passwortes. Je größer die Anzahl, desto mehr Passwörter müssen *im Durchschnitt* ausprobiert werden, um das Passwort zu erraten – und umso kleiner ist die Wahrscheinlichkeit, dass Passwort innerhalb einer bestimmten Anzahl an Versuchen zu erraten (sei es, weil man z. B. nur 3 Versuche hat, oder sei es, weil man ab einer bestimmten Anzahl einfach Jahre braucht, um alle durchzuprobieren).

Ein anderes und mathematisch etwas flexibleres Maß für die Stärke eines Passworts ist die Shannon-Entropie[1]. Sie gibt einfach gesagt an, wie vielen rein zufälligen Bits die Zufälligkeit eines Passwortes entspricht. Ein Vorteil der Shannon-Entropie ist, dass sie linear in der Länge des Passwortes ist. Ein einziges zufälliges Zeichen aus der oben genannten Menge mit 72 Zeichen hat eine Shannon-Entropie $H_{\text{Zeichen}} \approx 6{,}16$ Bit.[2] Ein Passwort aus 11 zufälligen Zeichen hat also eine Shannon-Entropie von

$$H_{11\text{ Zeichen}} \approx 67{,}9\text{ Bit}$$

Was ist nun die Entropie eine unserer Silben aus der Lösungsidee? Mit einer Wahrscheinlichkeit von $\frac{1}{4}$ wählen wir je

a) eine Ziffer zwischen 0 und 99 (100 Möglichkeiten),

b) einen Konsonanten und einen Vokal (210 Möglichkeiten)[3],

c) einen Vokal und einen Konsonanten (wieder 210 Möglichkeiten) oder

d) einen Konsonanten, einen Vokal und einen Konsonanten (4410 Möglichkeiten).

Damit erhalten wir für eine Silbe, die nach diesem Schema generiert wurde, eine Shannon-Entropie von:[4]

$$H_{\text{Silbe}} \approx 10{,}54\text{ Bit}$$

Um ein Passwort zu erhalten, das etwa genauso sicher ist wie das ursprüngliche Passwort, müssten wir also ein Passwort mit sechs bis sieben Silben generieren:

$$H_{7\text{ Silben}} = 7 \cdot H_{\text{Silbe}} \approx 73{,}8\text{ Bit}$$

Das ist schon einigermaßen lang, aber immer noch viel leichter zu merken als 11 rein zufällige Zeichen.

Moderne Empfehlungen zum Generieren sicherer Passwörter gehen inzwischen dazu über, Passwörter nicht mehr aus Silben zu bilden, damit sie lediglich *aussprechbar* werden, sondern sie stattdessen gleich aus ganzen Wörtern zu bilden. Wichtig ist dabei aber, dass die Wörter nicht von Menschen ausgewählt, sondern wirklich zufällig aus einer großen Liste von Wörtern gewählt werden. Damit lassen sich bei gleichbleibender Entropie wesentlich leichter zu merkende Passwörter generieren. Das liegt daran, dass sich aus den Wörtern oft Geschichten bilden lassen – oftmals absurde und lustige, aber dadurch auch besonders gut zu merken! Besonders verbreitete Bekanntheit hat dieses Verfahren mit einem Cartoon von Randall Munroe erlangt,

[1] `https://de.wikipedia.org/wiki/Entropie_(Informationstheorie)`

[2] $H_{\text{Zeichen}} = -\sum_{i=1}^{72} \frac{1}{72} \log_2 \frac{1}{72} = -\log_2 \frac{1}{72}$

[3] Wir gehen in dieser Berechnung von 10 Vokal-Phonemen und 21 Konsonanten aus, s. Abschnitt J1.2.

[4] $H_{\text{Silbe}} = -\sum_{i=1}^{100} \frac{1}{4\cdot 100} \log_2 \frac{1}{4\cdot 100} - \sum_{i=1}^{210} \frac{1}{4\cdot 210} \log_2 \frac{1}{4\cdot 210} - \sum_{i=1}^{210} \frac{1}{4\cdot 210} \log_2 \frac{1}{4\cdot 210} - \sum_{i=1}^{4410} \frac{1}{4\cdot 4410} \log_2 \frac{1}{4\cdot 4410}$
$= -\frac{1}{4}\left(\log_2 \frac{1}{4\cdot 100} + \log_2 \frac{1}{4\cdot 210} + \log_2 \frac{1}{4\cdot 210} + \log_2 \frac{1}{4\cdot 4410}\right)$

der in Abbildung J1.1 abgedruckt ist. Auch hier verwendet Munroe die Shannon-Entropie als wesentliches Maß für die Stärke von Passwörtern.

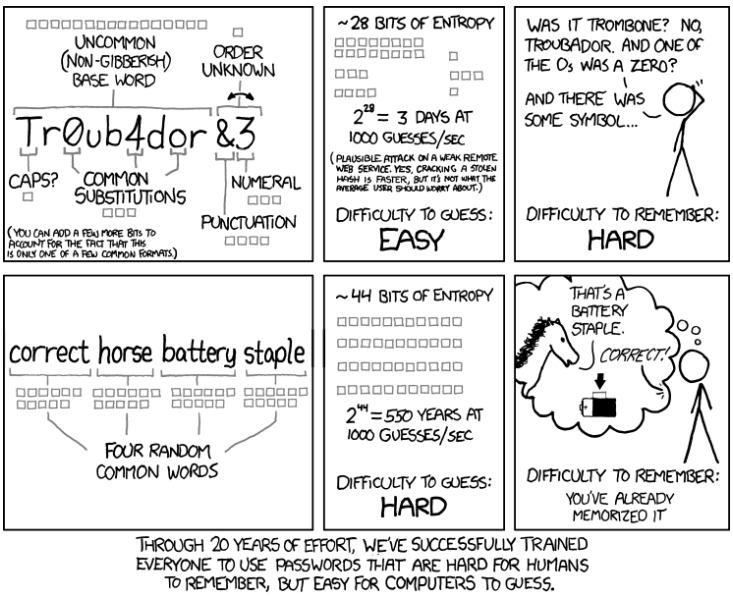



Abbildung J1.1: Cartoon von Randall Munroe zur Sicherheit von Passwörtern. Heruntergeladen von `https://xkcd.com/936/`. Titeltext für Browser: „To anyone who understands information theory and security and is in an infuriating argument with someone who does not (possibly involving mixed case), I sincerely apologize.“

## J1.4 Beispiele

### Programmablauf

Im Normalfall zeigt das Programm keine Zwischenschritte an. Beispielhaft wird daher hier ein Blick auf die interne Verarbeitung geworfen.
Im ersten Schritt wird die Anzahl an Silben (**length**) auf 5 gesetzt.
Die Silbentypen (**syllableTypes**) sind: 3, 2, 3, 1, 1
Damit ist die Silbenstruktur (**passwordStructure**): N, CVC, N, CV, CV
Nach **GenerateLetters** ergibt sich daraus: 65rod3fusei

Die Struktur **N** wurde ein Mal durch eine Ziffer und ein anderes Mal durch zwei Ziffern ersetzt, ebenso das letzte **V** durch den Doppellaut **ei**. Das erzeugte Passwort kann man durchaus aussprechen.

### Ausgaben

Hier einige subjektiv lustige Programmausgaben, jeweils mit ihrer Zerlegung in Silben und der zugehörigen Silbenstruktur.

- beariefbuschhaschu
  `bie-ar-ief-busch-ha-schu`
  `CV -VC- VC-CVC  -CV- CV`

- titlideitujeieidquie
  `tit-li-dei-tu-jei-eid-quie`
  `CVC-CV-CV -CV-CV - VC- CV`
- du98mehubschir8quu
  `du-98-me-hub-schir-8-quu`
  `CV-N -CV-CVC-  CVC-N-C V`
- nussiemfusdimienviewiel
  `nus-siem-fus-di-mien-vie-wiel`
  `CVC-CV C-CVC-CV-CV C-CV -CV C`
- mameworielpeg
  `ma-me-wo-riel-peg`
  `CV-CV-CV-CV C-CVC`
- kenschu28teyllalof
  `ken-schu-28-teyl-la-lof`
  `CVC-  CV-N -CV C-CV-CVC`

## J1.5 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Passwörter nicht gut auszusprechen**
  Ob explizite Regeln oder ein Erzeugungssystem mit impliziten Regeln wie in dieser Beispiellösung: Die Aufgabenstellung fordert auf jeden Fall, dass die Passwörter „einigermaßen gut auszusprechen und deshalb leichter zu merken sind“. Ob diese Forderung erfüllt ist, mag Geschmackssache sein. Ein rein zufällige Aneinanderreihung von Zeichen ist aber sicher nicht akzeptabel. Auch andere Verfahren, die sich ungenügend am Aussprechen orientieren, können zu Punktabzug führen. Es soll zumindest einigermaßen sinnvoll begründet sein, warum und wie das Verfahren zur Erzeugung von Passwörtern sich an der Aussprechbarkeit orientiert.
- $[-1]$ **Passwörter zu leicht zu erraten**
  Die Aufgabenstellung fordert auch, dass die Passwörter schwer zu erraten sind. Werden regelmäßig Passwörter erzeugt, die aus weniger als 10 Zeichen bestehen, ist diese Forderung nicht erfüllt. Wenn die Länge des Passworts im Programm frei einstellbar ist, ist das in Ordnung; es sei denn, in der Dokumentation werden zu kurze Passwörter präsentiert. Eine Diskussion der Passwortstärke war nicht erwartet.
- $[-1]$ **Implementierung fehlerhaft oder unnötig aufwendig**
  Die Implementierung sollte die Regeln oder das Erzeugungsverfahren wie beschrieben umsetzen. Die Generierung der Passwörter sollte nicht unnötig kompliziert realisiert werden.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Es sollten mindestens drei erzeugte Passwörter angegeben werden.

# Junioraufgabe 2: Baulwürfe

## J2.1 Lösungsidee

### Darstellung der Maulwurfshügel

Unser Programm bekommt als Eingabe eine Karte der Höhe $h$ und Breite $b$. Die Karte der Forscher kann als zweidimensionales boolesches Array $A$ der Größe $h \times b$ dargestellt werden. Wir lesen unsere Eingabe so ein, dass in $A[i][j]$ der boolesche Wert `true` steht, wenn in der Karte der Forscher in der $i$-ten Zeile und der $j$-ten Spalte ein Hügel verzeichnet ist.

### Eine Erste Lösungsidee

Eine der ersten Lösungsideen, die sich beim Durchlesen der Aufgabenstellung ergibt, ist die folgende: Überprüfe für jedes Planquadrat, ob es die linke obere Ecke eines Baulwurfshügels darstellt. Das Programm in Pseudocode sieht also folgendermaßen aus:

```
 1: procedure BAULWURFSBAUE(A, h, b)
 2:     baulwurfsbaue ← 0
 3:     for z ← 0,..,h−4 do                                  ▷ Zeilenindex
 4:         for s ← 0,..,b−3 do                              ▷ Spaltenindex
 5:             if A[z][s] and A[z][s+1] and A[z][s+2] and
 6:               A[z+1][s] and not A[z+1][s+1] and A[z+1][s+2] and
 7:               A[z+2][s] and not A[z+2][s+1] and A[z+2][s+2] and
 8:               A[z+3][s] and A[z+3][s+1] and A[z+3][s+2] then
 9:                 baulwurfsbaue ← baulwurfsbaue + 1
10:             end if
11:         end for
12:     end for
13:     return baulwurfsbaue
14: end procedure
```

Aber Achtung! Diese Methode hat noch ein Problem. Schauen wir uns einen Ausschnitt aus der Beispielkarte 5 an:

Wie die Forscher schnell herausfinden können, gibt es auf dieser Karte nur 4 Baulwurfsbaue. Trotzdem findet der oben beschriebene Algorithmus 5 Baue. Warum ist das der Fall? Neben

den 4 „echten“ Bauen (Abbildung J2.1) gibt es auf der Karte ein fünftes Mal die gesuchte Struktur (Abbildung J2.2). Weil sich aber, wie die Aufgabenstellung sagt, Baulwurfsbaue nicht überlappen, kann es sich hier nicht um einen Bau handeln.

Abbildung J2.1: Die 4 Baulwurfsbaue

Abbildung J2.2: Ein Extrabau[5]

Unser Programm findet hier also alle wirklichen Baulwurfsbaue, erkennt allerdings auch noch eine weitere Kombination an Hügeln als Baulwurfsbau. Die Lösung muss also noch ein bisschen angepasst werden, um dieses Problem zu verhindern. Eine Möglichkeit zur Anpassung ist beispielsweise die folgende: Sobald ein Baulwurfsbau gefunden wird, markiere jeden Hügel, der bereits in diesem Bau vorkommt. Wird nun ein neuer Baulwurfsbau gefunden, überprüfe zuerst, ob in diesem Bau bereits verwendete Hügel vorkommen. Nur wenn dies nicht der Fall ist, wird der neu gefundene Bau gezählt.

Mit dieser Lösung kann es natürlich passieren, dass in unserem Beispiel nur ein Baulwurfsbau gefunden wird, wenn der Bau in der Mitte zuerst gefunden wird. Dies kann allerdings nicht passieren, wenn beispielsweise alle Felder von links nach rechts und oben nach unten durchsucht werden. Startet man allerdings in der Mitte mit der Suche, kann dieser Ansatz falsche Ergebnisse liefern.

### Einfachere Lösung

Die Aufgabenstellung macht noch eine weitere Vorgabe: Ein Planquadrat mit einem Maulwurfshügel ist immer rundum von Quadraten ohne Hügel umgeben. Damit kann die folgende Beobachtung angestellt werden: Sobald ein Quadrat mit Hügel an ein anderes Quadrat mit Hügel angrenzt, muss es automatisch ein Planquadrat mit einem Baulwurf sein! Eine weitere Beobachtung hilft unserem Lösungsansatz noch weiter: Jeder Hügel, welcher von Baulwürfen gebaut wird, grenzt in mindestens einer Richtung (oben, unten, links, rechts) an ein weiteres Planquadrat mit Hügel an!

Daher kann für jeden Hügel einfach überprüft werden, ob von diesem aus nach oben, unten, links oder rechts mindestens ein weiterer Hügel verzeichnet ist. Wenn ja, wird der Hügel als *Baulwurfshügel* markiert. Dieses System findet alle Hügel, die von Baulwürfen (als Teil ihrer Baue) gebaut wurden, und keinen weiteren Hügel.

Nun wissen wir, dass ein Baulwurfsbau immer die folgende Struktur besitzt:

[5]Es ist nicht möglich, dass auf der ganzen Karte nur der rot markierte Baulwurfsbau zu finden ist. Wäre dies

```
XXX
X X
X X
XXX
```

Damit besteht ein Baulwurfsbau immer aus genau zehn Baulwurfshügeln. Die Anzahl an Baulwurfsbauen ist also einfach die Anzahl der Baulwurfshügel geteilt durch zehn.

In Pseudocode sieht unser Algorithmus also nun wie folgt aus:

```
1:  procedure BAULWURFSBAUE(A, h, b)
2:      baulwurfshuegel ← 0
3:      for z ← 0,..,h−1 do                                   ▷ Zeilenindex
4:          for s ← 0,..,b−1 do                               ▷ Spaltenindex
5:              if A[z][s] then
6:                  if A[z−1][s] or A[z][s+1] or A[z+1][s] or A[z][s−1] then
7:                      baulwurfshuegel ← baulwurfshuegel + 1
8:                  end if
9:              end if
10:         end for
11:     end for
12:     return baulwurfshuegel ÷ 10
13: end procedure
```

In diesem Code wird an einigen Stellen auf das Array an Stellen zugegriffen, die außerhalb des Kartenbereichs liegen. Dies kann zu Fehlern im Programm führen. In unserem Pseudocode wird immer erwartet, dass an Stellen außerhalb des Kartenbereichs der Wert `false` in $A$ steht.

#### Variante

Genauso gut kann natürlich die Anzahl an Maulwurfshügeln gezählt werden, indem überprüft wird, dass kein angrenzendes Quadrat einen Hügel aufweist. Die Anzahl der Baulwurfsbaue ergibt sich dann als

$$\frac{\text{Gesamtanzahl Hügel - Maulwurfshügel}}{10}$$

## J2.2 Beispiele

Angewandt auf die von BWINF bereitgestellten Eingaben liefert unser Programm die folgenden Ergebnisse:

der Fall, sind alle anderen schwarzen `X` normale Maulwürfe. Allerdings wissen wir, dass Maulwürfe ihre Baue nicht aneinander angrenzend bauen.

| Eingabe | **Ergebnis** | Ergebnis einfache Lösung |
|---|---|---|
| `karte0.txt` | **3** | 3 |
| `karte1.txt` | **37** | 37 |
| `karte2.txt` | **32** | 32 |
| `karte3.txt` | **318** | 318 |
| `karte4.txt` | **3189** | 3193 |
| `karte5.txt` | **16** | 20 |
| `karte6.txt` | **559** | 559 |

## J2.3 Bewertungskriterien

Die Bewertungskriterien (Fettdruck) vom Bewertungsbogen werden hier näher erläutert (Punktabzug in []).

- [−1] **Modellierung ungeeignet**
  Die offensichtliche Möglichkeit zur Repräsentation der Hügelkarte ist ein zweidimensionales Array. Ob boolesche Werte, die Zeichen der Eingabe (Leerzeichen und X) oder ein anderes Wertepaar verwendet werden: das ist alles gut brauchbar. Darstellungen, die das Lösungsverfahren unnötig verkomplizieren, führen zum Punktabzug.
- [−1] **Lösungsverfahren fehlerhaft**
  Die Aussage der Aufgabenstellung, dass sich Baulwurfsbaue nicht überlappen, konnte auch als Versprechen verstanden werden, dass sich in der Eingabe auch keine Baulfwurfsbaue so berühren, dass es zu scheinbaren Überlappungen der Baulfwurfsbau-Strukturen kommt. In diesem Fall funktioniert auch das zuerst beschriebene, einfache Verfahren. Es wird deshalb nicht verlangt, dass das Verfahren scheinbar überlappende Strukturen erkennt und ausschließt. Damit sind auch die Werte in der rechten Spalte der Ergebnistabelle akzeptabel. Ansonsten sollten Baulwurfsbaue aber zuverlässig erkannt werden, auch wenn sie aneinander angrenzen oder am Rand der Karte liegen. Abgesehen von den scheinbar überlappenden Strukturen dürfen aber natürlich auch nicht mehr Baue erkannt werden als vorhanden sind.
- [−1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Nur besonders umständliche Verfahren bzw. Implementierungen führen zu Punktabzug. Das Programm sollte in der Lage sein, alle BWINF-Beispiele in sehr kurzer Zeit zu lösen.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Es sollten die Ausgaben zu mindestens drei der sechs zusätzlichen Beispiele zu der Aufgabe (`karte1.txt` bis `karte6.txt`) in der Dokumentation abgedruckt sein.

# Aufgabe 1: Wörter aufräumen

## 1.1 Lösungsidee

Den Lückentext auszufüllen bedeutet nichts anderes, als eine Zuordnung zu finden, die jedem Wort aus dem Lückentext (das wir auch *Lücke* nennen) genau ein passendes Wort aus der Wortliste zuweist. Dabei dürfen die Wörter aus der Wortliste nicht doppelt verwendet werden; die Zuordnung ist also „eineindeutig“.

Die Lösung lässt sich daher in drei Teilschritte aufteilen:

1. Erkennen der Lücken (bzw. Aufteilen des Lückentexts in Lücken)
2. Finden der passenden Wörter für jede Lücke
3. Finden einer Zuordnung zwischen Lücken und passenden Wörtern, bei der jedes Wort genau einer Lücke zugeordnet wird.

Es ist sinnvoll, sowohl die Lücken als auch die Wörter in je einer Liste zu speichern, damit sie später über den Index (also die Position in der Liste, beginnend mit 0) eindeutig erkannt werden können.

#### Aufteilen des Lückentexts

Das Auftrennen des Lückentexts in einzelne Lücken ist einfach: Man kann den Lückentext immer an Leerzeichen trennen. Die meisten Programmiersprachen stellen hierfür eine Funktion zur Verfügung.

Etwas schwerer ist die Behandlung von Satzzeichen, da diese nicht verloren gehen sollen. Da in den Beispielen nur Satzzeichen auftreten, die am Ende eines Wortes stehen, ist z. B. eine Möglichkeit, diese einfach stehen zu lassen und beim Zuordnen der Wörter zu ignorieren. Dann muss man aber bei der Ausgabe der Lösung am Ende bei jedem Wort kontrollieren, ob am Ende der Lücke ein Satzzeichen steht.

#### Finden der passenden Wörter

Jede Lücke muss mit jedem Wort verglichen werden. Wenn dabei am Ende der Lücke ein Satzzeichen steht, muss dieses (für den Vergleich) entfernt werden. Wenn das Wort nicht die gleiche Länge hat wie die Lücke, kann der Vergleich sofort abgebrochen werden; das Wort passt nicht. Ansonsten wird für jedes Zeichen im Wort kontrolliert, ob die Lücke an der gleichen Stelle das selbe Zeichen oder eine Leerstelle (also einen Unterstrich) hat. Ist dies nicht der Fall, so gibt die Lücke an der Stelle ein anderes Zeichen vor und das Wort passt nicht.

Für jede Lücke wird eine Liste angelegt, in der die Indizes der passenden Wörter stehen.

#### Finden einer Verteilung

#### Einfache Lösung: Backtracking

Die Zuordnung kann mithilfe von Backtracking gefunden werden: In einer Liste wird für jedes Wort gespeichert, ob es bereits verwendet wurde oder nicht. Eine rekursive Funktion bekommt

den Index (also die Nummer) der Lücke übergeben, die sie aktuell füllen soll. Für diese Lücke probiert sie nacheinander alle passenden Wörter aus: Wenn ein Wort noch nicht verwendet wurde, wird es als verwendet markiert und die Funktion mit dem nächsten Index aufgerufen. Anschließend wird das Wort wieder als nicht verwendet markiert und das nächste Wort ausprobiert. Wörter, die aktuell als verwendet markiert sind, werden übersprungen.

Wenn die Funktion mit einem Index aufgerufen wird, der größer oder gleich der Anzahl Lücken ist, so ist jeder Lücke ein Wort zugeordnet worden und die aktuelle Zuordnung kann als Lösung ausgegeben werden.

#### Schnellere Lösung: „verbessernde Wege“

Zu einer schnelleren Lösung kommt man, indem man die Lücken und Wörter als Knoten in einem Graphen darstellt. Wenn ein Wort zu einer Lücke passt, dann existiert zwischen den entsprechenden Knoten eine Kante.

Das Problem, das wir lösen wollen, nennt sich *bipartites Matching*[6]: Wir haben einen Graphen mit zwei disjunkten, also völlig verschiedenen Knotenmengen (bipartit) – die Lücken und die Wörter – und möchten zwischen diesen möglichst viele[7] Paare bilden (Matching), die je aus einer Lücke und einem passenden Wort bestehen.

Die folgende Lösung beruht auf einer Modellierung mithilfe von Flüssen[8].

Grob vereinfacht bedeutet das, dass wir möglichst viele Wege von Lücken zu Wörtern suchen, wobei wir erlauben, bereits verwendete Kanten rückwärts zu verwenden: Wenn wir eine Kante verwenden, die zu einem bereits verwendeten Wort führt, folgen wir der zuvor verwendeten Kante zurück zu der Lücke, wo dieses Wort verwendet wurde, und suchen für diese Lücke ein neues passendes Wort.

Dazu beginnt man mit einem Zustand, in dem es noch keine Paare gibt, und sucht nacheinander von jeder Lücke aus einen *verbessernden Weg* (engl.: augmenting path), also einen Weg durch den Graphen, der zu insgesamt einem Paar mehr führt.

Diese Pfade findet man mittels einer Tiefensuche: Gelangt man von einer Lücke zu einem noch nicht verwendeten Wort, so hat man ein neues Paar und damit einen verbessernden Weg gefunden. Gelangt man stattdessen zu einem bereits verwendeten Wort, so sucht man von dessen bisherigem Partner aus nach einem verbessernden Weg: Wenn dieser existiert, kann die Lücke auch mit einem anderen Wort gefüllt werden, und man hat ein neues Paar gefunden.

Da von jeder Lücke aus eine solche Tiefensuche ausgeführt wird, die im schlimmsten Fall alle Kanten einmal besucht, liegt die Laufzeit bei diesem Ansatz in $O(|V| \cdot |E|)$ (also auch in $O(|V|^3)$, da der Graph bipartit ist). Das Aufbauen des Graphen im vorigen Schritt braucht allerdings noch weitere $O(|V|^2 \cdot m)$ Zeit, wobei $m$ die maximale Wortlänge ist.

## 1.2 Beispiele

Hier sind die Lösungen für die vorgegebenen Lückentexte:

[6] https://de.wikipedia.org/wiki/Matching_(Graphentheorie)

[7] in diesem Fall genau so viele wie Lücken, aber allgemein kann das Problem auch für unterschiedlich große Mengen gelöst werden

[8] https://de.wikipedia.org/wiki/Flüsse_und_Schnitte_in_Netzwerken

**raetsel0.txt** oh je, was für eine arbeit!

**raetsel1.txt** Am Anfang wurde das Universum erschaffen. Das machte viele Leute sehr wütend und wurde allenthalben als Schritt in die falsche Richtung angesehen.

**raetsel2.txt** Als Gregor Samsa eines Morgens aus unruhigen Träumen erwachte, fand er sich in seinem Bett zu einem ungeheueren Ungeziefer verwandelt.

**raetsel3.txt** Informatik ist die Wissenschaft von der systematischen Darstellung, Speicherung, Verarbeitung und Übertragung von Informationen, besonders der automatischen Verarbeitung mit Digitalrechnern.

**raetsel4.txt** Opa Jürgen blättert in einer Zeitschrift aus der Apotheke und findet ein Rätsel. Es ist eine Liste von Wörtern gegeben, die in die richtige Reihenfolge gebracht werden sollen, so dass sie eine lustige Geschichte ergeben. Leerzeichen und Satzzeichen sowie einige Buchstaben sind schon vorgegeben.

## 1.3 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert (Punktabzug in []).

- $[-1]$ **Interne Darstellung ungeeignet**
  Sowohl die Wortliste als auch der Lückentext können gut als Liste, eindimensionales Array oder Ähnliches verwaltet werden. Akzeptabel ist auch, wenn mit Strings und Indizes darin gearbeitet wird. Der Zugriff auf Wörter und Lücken sollte aber die Implementierung weder verkomplizieren noch verlangsamen. Umlaute und 'ß' sollten jeweils als ein Buchstabe behandelt werden, auch wenn sie im Zeichensatz zwei Bytes belegen (korrekte Unicode-Behandlung); in der Regel werden die Mechanismen der Programmiersprache dafür sorgen.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das Lösungsverfahren darf nicht einfach einer Lücke das erste passende Wort fest zuordnen (greedy). So können nicht zuverlässig beliebige Lückentexte vervollständigt werden. Auch andere Mängel, die zu falschen Ergebnissen führen, sind nicht akzeptabel. Da die Aufgabenstellung eindeutige richtige Lösungen garantiert, ist es nicht erforderlich, mehrere Lösungen verwalten zu können.
- $[-1]$ **Satzzeichen nicht richtig behandelt**
  Repräsentation und Lösungsverfahren sollten mit den Satzzeichen korrekt umgehen.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Die Lösungssuche soll nicht wesentlich aufwendiger sein als nötig. Untern anderem soll möglichst schnell erkannt werden, dass ein Wort nicht zu einer Lücke passt.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Es sollten Lösungen zu mindestens zwei der vier zusätzlichen Beispiele (`raetsel1.txt` bis `raetsel4.txt`) zu der Aufgabe in der Dokumentation abgedruckt sein.

# Aufgabe 2: Dreieckspuzzle

## 2.1 Lösungsidee

Als erstes kann man sich die Anzahl der Möglichkeiten anschauen, die es gibt, die Teile des Puzzles anzuordnen. In der Aufgabe hat Lizzy 9 dreieckige Puzzleteile, die sie auf die 9 Positionen im großen Dreieck verteilen will. Für das erste Teil gibt es 9 mögliche Positionen, für das nächste dann nur noch 8, danach 7 und so weiter. Folglich gibt es

$$9 \cdot 8 \cdot 7 \cdot ... \cdot 2 \cdot 1 = 9! = 362\,880$$

mögliche Anordnungen. Dabei haben wir aber einen wichtigen Aspekt außer Acht gelassen: Jedes Puzzleteil lässt sich auf 3 verschiedene Weisen auf eine Position im Puzzle platzieren, nämlich jeweils um 60° gedreht. Da wir 9 Teile haben, die sich in je 3 verschiedenen Drehwinkeln im Puzzle befinden können, haben wir zusätzlich zur Anordnung noch

$$3 \cdot 3 \cdot \cdots \cdot 3 = 3^9 = 19683$$

verschiedene Möglichkeiten. Zusammen ergibt das

$$9! \cdot 3^9 = 7\,142\,567\,040$$

Möglichkeiten, die Dreiecke zu einem Dreieckspuzzle zusammenzufügen. Für ein Puzzle dieser Größe ist es daher schwierig, wenn auch machbar, alle Anordnungen durchzugehen, um eine Lösung zu finden – also eine Anordnung, in der alle Figurenhälften zueinander passen. Viele Anordnungen erweisen sich allerdings schon nach der Platzierung weniger Puzzleteile als falsch: Sobald wir eine Kante haben, an der zwei nicht passende Figurenhälften aufeinander stoßen, können wir aufhören.

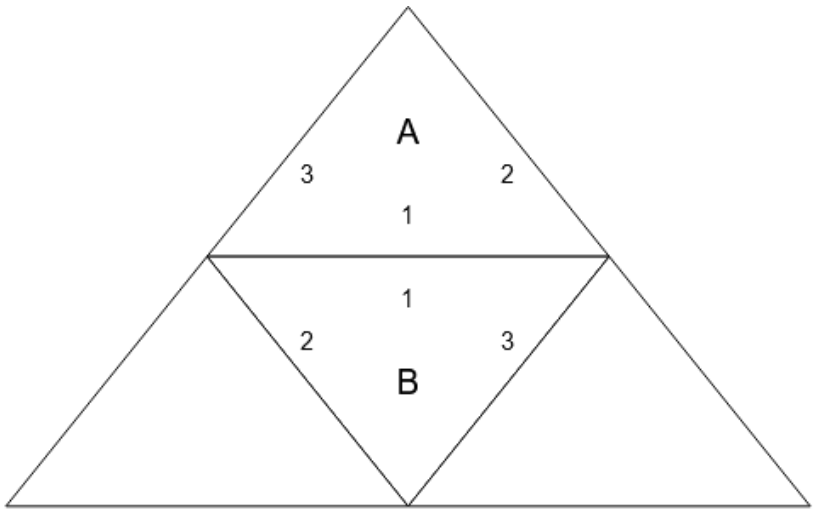
### Rekursives Backtracking

Die Lösungsidee lässt sich als Backtracking-Algorithmus umsetzen. Dabei werden solange Puzzleteile zu einer Anordnung hinzugefügt, bis das Puzzle fertig ist oder es keine Möglichkeit mehr gibt, weitere Teile anzufügen. Im ersten Fall sind wir fertig, im zweiten Fall geht der Algorithmus einen Schritt zurück und versucht ein anderes Puzzleteil. Dabei werden nur solche Teile hinzugefügt, die passende Figuren haben. Dieses Verfahren ist wesentlich effizienter; es stellt sich letztlich heraus, dass nur wenige tausend Iterationen notwendig sind, um entweder eine Lösung zu finden oder auszuschließen, dass es eine gibt.

Um das Puzzle mittels eines Backtracking-Algorithmus zu lösen, muss zunächst das Problem modelliert werden. Die Puzzleteile liegen als Tupel aus drei Zahlen vor, und es bietet sich an, diese Repräsentation zu behalten. Dabei soll die Reihenfolge der Zahlen den Winkel mit repräsentieren. Die beiden Tupel $(1,2,3)$ und $(2,3,1)$ stellen also das selbe Puzzleteil da, nur anders gedreht. In Abbildung 2.1 ist eine beispielhafte Aufteilung zu sehen. Es gibt zwei verschiedene Dreiecke: solche, die auf einer Seite stehen (A), und solche, die auf einer Spitze stehen (B). Die Zahlen an den Seiten zeigen die Position im Tupel an, an der der Wert für die entsprechende Figurenhälfte gespeichert wird.

Die Dreiecke werden von links nach rechts und dann von unten nach oben nummeriert. Dies erlaubt es uns die Kanten in einer Puzzle-Anordnung, an denen sich die Puzzleteile berühren,

Abbildung 2.1: Nummerierung der Kanten




zu beschreiben. Damit ist gemeint, dass wir wissen, welche Seiten zweier Teile an welcher Stelle (hier auch Kante genannt) der Anordnung aneinander angelegt sind. In jeder Reihe der Anordnung kommen zwei neue Dreiecke hinzu. Insgesamt führt dies dazu, dass die Anzahl der Dreiecke im Puzzle immer eine Quadratzahl sein muss, da folgende Formel gilt[9]:

$$\sum_{k=0}^{n-1}(2*k+1) = n^2$$

Die Aufgabe fordert nur, dass Dreieckspuzzle mit 9 Teilen gelöst werden müssen. Daher wäre es möglich, alle Kanten statisch anzugeben. Dennoch sei hier ein Algorithmus gegeben, der es ermöglicht, zu einem beliebigen Dreieckspuzzle die Kanten zu finden. Das geschieht einmal initial vor der Ausführung des eigentlichen Algorithmus zur Lösungsbestimmung. Dies ist bei der Implementierung des Backtracking-Algorithmus zu beachten. Zusätzlich wird zu jeder Kante die Position der Seite der Figur im Tupel gespeichert. Diese ist für beide Teile gleich, siehe Abbildung 2.1. Wir unterscheiden zwischen Kanten auf einer horizontalen Ebene und Kanten auf der vertikalen. Die horizontalen Kanten gehen immer zum direkten Nachbarn (das nachfolgende Puzzleteil im Array), die Berechnung der vertikalen ist etwas komplizierter. In jeder Reihe kommen zwei neue Dreiecke hinzu. Der Algorithmus merkt sich die Länge der aktuellen Reihe und berechnet so das darunter liegende Dreieck.

Zum Beispiel besitzt das nullte Puzzleteil – das oberste – die Kante $(2,0)$; es hat also eine gemeinsame Kante mit dem zweiten Puzzleteil (das unter dem obersten), und es berühren sich die 0-ten Seiten. Das Ergebnis ist dann ein 2-dimensionaler Kanten-Array, der für jedes Puzzleteil eine Liste an gemeinsamen Kanten hat. Wenn die Figuren aus der Sicht eines Puzzleteils stimmen, dann folgt natürlich, dass diese Figuren auch andersherum zusammenpassen. Deswegen wird jede Kante nur einmal gespeichert. So hat also das Puzzleteil unten rechts gar keine Kanten mehr, da diese schon in den Kantenmengen für die daneben und darüber liegenden Teile gespeichert wurden. Zusammenfassend sieht der Kanten-Algorithmus so aus:

[9] Beweis durch z.B. vollständige Induktion oder grafische Anschauung

**Algorithmus 1** Kanten finden

**Input:** Anzahl der Puzzleteile $n$
**Output:** Kantenmenge

```
1:  Kanten = [][]          ▷ Initialisiere Kantenmenge für alle Puzzleteile zu einer leeren Liste
2:  index = 0
3:  schritt = 1
4:  repeat √(Anzahl Puzzleteile)
5:      for i ← index to index + schritt − 1 do
6:          Füge Kante (i + 1, i + (i + 1)%2) zu Kanten[i] hinzu
7:
8:      if index + schritt < n then
9:          for i ← index to i + schritt + 1 in 2er-Schritten do
10:             Füge Kante (i + schritt + 1, 0) zu Kanten[i] hinzu
11:     end if
12:
13:     index += schritt
14:     schritt += 2
15: return Kanten
```

Ein weiterer wichtiger Schritt beim Backtracking ist, dass die Konsistenz der aktuellen Anordnung überprüft wird. Das ist nötig, um passende Teile zu finden, mit der die aktuelle Anordnung erweitert wird. Konkret heißt das hier, dass die Figuren aller schon gesetzten Puzzleteile an den Kanten stimmig sind. Dafür iteriert der Algorithmus über alle bereits vorhandenen Kanten, also Kanten an denen beide Puzzleteile schon gesetzt wurden, und schaut ob die Summe der anliegenden Figuren 0 ergibt, da eine negative Zahl für die Unterseite und eine positive für die Oberseite einer Figur steht. Der Algorithmus sei hier gegeben:

**Algorithmus 2** Konsistenz der Anordnung überprüfen

**Input:** Aktuelle Anordnung, Kanten
**Output:** True wenn kosistent, sonst False

```
1: for each Position in Puzzle, die bereits belegt wurde
2:     for each Andere Position, Drehwinkel in Kanten[Position]
3:         if Anordnung[Teil][Position] − Anordnung[Anderes Teil][Position] ≠ 0 then
4:             return False
5:         end if
6: return True
```

Der Backtracking-Algorithmus ruft sich rekursiv wieder selbst auf, er gibt entweder eine richtige Anordnung der Puzzleteile oder Falsch zurück, um zu signalisieren, dass es keine gibt. Dabei wird jedes Puzzlestück für jeden möglichen Drehwinkel ausprobiert. Sobald der Backtracking-Algorithmus für ein Puzzleteil ausgibt, dass es keine möglichen Folgekonfigurationen geben kann, wird das nächste Puzzleteil ausprobiert.

**Algorithmus 3** Backtrack()

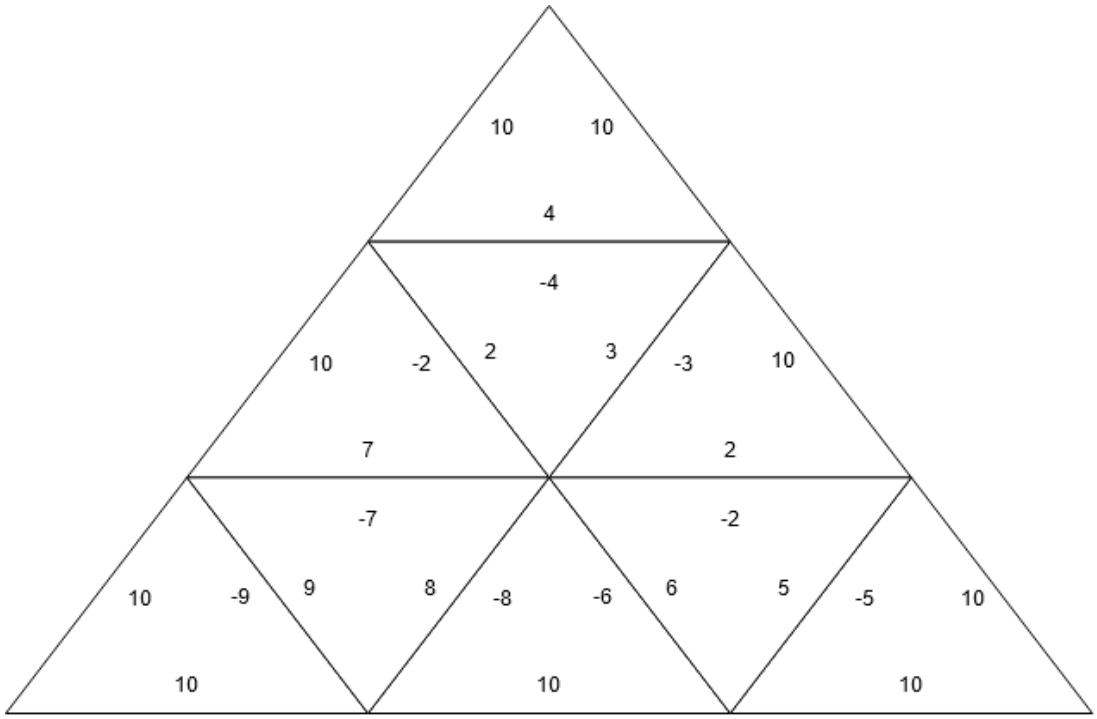
**Input:** Anordnung, verbleibende Puzzleteile
**Output:** gelöstes Puzzle oder Falsch, wenn unlösbar

```
1:  if alle Puzzleteile plaziert then
2:      return Puzzle
3:  end if
4:  nächstesTeil ← nächste aufsteigend freie unbelegte Position
5:  for each noch nicht plaziertes Puzzleteil
6:      for each möglichen Winkel des Puzzleteils
7:          if neues Teil passt / neues Puzzle konsistent then
8:              setzte Teil ins Puzzle ein
9:              Rest ← verbleibende Puzzleteile
10:             ergebnis ← Backtrack(Anordnung, Rest)
11:             if ergebnis then
12:                 return Puzzle
13:             end if
14:         end if
15: return False
```

## 2.2 Beispiele

Von den vier gegebenen Beispielen sind alle lösbar, exemplarisch sei hier eine Lösung für das vierte Puzzle gezeigt (`puzzle3.txt`).

Abbildung 2.2: Beispiel Lösung puzzle3.txt



## 2.3 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- [−1] **Interne Darstellung umständlich / ungeeignet**
  Die interne Darstellung der Teile sollte unter anderem ein einfaches Drehen der Teile erlauben. Die interne Verwaltung der Anordnungen sollte die Positionen der Teile im Puzzle und die Beziehungen zwischen den Positionen geeignet verwalten. Insbesondere sollte die Prüfung, ob zwei Teile bzw. alle Teile einer Anordnung zueinander passen, leicht durchzuführen sein. Es ist in Ordnung, wenn die Modellierung der Anordnungen auf ein Dreieckspuzzle mit neun Teilen fest zugeschnitten ist.
- [−1] **Lösungsverfahren fehlerhaft**
  Das Verfahren muss wie gefordert arbeiten, also zum einen korrekt entscheiden, ob es eine Lösung für die gegebenen Puzzleteile gibt und, falls ja, eine Anordnung der Teile ausgeben, bei der alle Figurenhälften zueinander passen.
- [−1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Die Lösungssuche soll nicht wesentlich aufwendiger sein als nötig. Insbesondere soll eine Teilanordnung nur passend erweitert werden. Eine uneingeschränkte Aufzählung aller beliebigen Anordnungen (auch solcher mit nicht zueinander passenden Seiten) ist nicht akzeptabel.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Die Ausgabe einer Lösung sollte idealerweise die Struktur des Puzzles abbilden, damit man direkt aus der Ausgabe die Gültigkeit der Lösung ablesen kann. Eine nachvollziehbare textuelle Ausgabe mit passenden Einrückungen ist ausreichend. Eine einfache Auflistung der Puzzleteile und ihres Rotationszustandes reicht nicht aus; es sei denn, diese Ausgaben werden durch manuell erzeugte Visualisierungen ergänzt.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Es sollten Lösungen für mindestens zwei der vier Beispiele zu der Aufgabe (`puzzle0.txt` bis `puzzle3.txt`) in der Dokumentation abgedruckt sein.

# Aufgabe 3: Tobis Turnier

## 3.1 Lösungsidee

Wir nummerieren die Spieler von 1 bis $n$ in der Reihenfolge der Eingabe.[10] In der Eingabe werden ihnen ganzzahlige Spielstärken $0 \leq a_1, \ldots, a_n \leq 100$ zugewiesen.

Ein Spiel zwischen Spielern $x$ und $y$ läuft – *effektiv* – so ab: Spieler $x$ legt $a_x$ rote und Spieler $y$ legt $a_y$ blaue Kugeln in eine gemeinsame Urne. Eine Kugel wird gezogen; ist sie rot, gewinnt Spieler $x$, ist sie blau, gewinnt Spieler $y$. Die Wahrscheinlichkeit, dass Spieler $x$ oder Spieler $y$ gewinnt, ist damit $p_x := P(x \text{ gewinnt}) = \frac{a_x}{a_x+a_y}$ bzw. $p_y = 1 - p_x = \frac{a_y}{a_x+a_y}$.[11]

Ein Turnier besteht aus einer Anzahl an Spielen, anhand deren Ausgang der Gewinner des Turniers bestimmt wird. Die genauen Regeln des Turniers hängen von der *Turnierform* ab und werden im Folgenden jeweils einzeln betrachtet.

Auf einen (unseren) prinzipiellen Lösungsweg zum Beantworten der in der Aufgabe gestellten Frage kann jedoch an dieser Stelle schon allgemein eingegangen werden. Ermittelt werden soll, „wie oft der spielstärkste Spieler im Durchschnitt über viele Wiederholungen des Turniers gewinnt“. Wir wählen einen direkten – den experimentellen – Ansatz und entwickeln ein Programm, das eine große Anzahl an Turnieren simuliert und ermittelt, welchen Anteil davon der/die spielstärkste Spieler(in) (der Einfachheit halber geben wir ihr hier den Namen Lina) gewonnen hat. Man kann sich das so vorstellen, als würden wir viele tatsächlich stattfindende Turniere beobachten und den Durchschnitt der Ergebnisse aufschreiben – nur, dass wir ein Programm schreiben, das die Turniere simuliert.

Die als Endergebnis gefragte Gewinnquote des spielstärksten Spielers ist dann die Anzahl der Turniersiege Linas geteilt durch die Anzahl der simulierten Turniere. Wir empfehlen Tobi diejenige Turniervariante, für die diese Quote am höchsten ist.

### Turnierform *Liga*

In diesem Fall spielt jeder Spieler gegen jeden anderen Spieler genau ein Mal. Der Spieler mit den meisten Siegen gewinnt das Turnier, bei einem Gleichstand der mit der kleineren Spielernummer. Die Umsetzung ist „straightforward“:

Nach dem Einlesen der Spielstärken und nachdem das Programm bestimmt hat, wer der stärkste Spieler Lina ist (sollte es mehrere Spieler mit der größten Spielstärke geben, wird der mit der kleinsten Nummer gewählt[12]), wird eine vorgegebene Zahl an Turnieren simuliert. Dazu wird für jedes Paar $1 \leq x < y \leq n$ zweier verschiedener Spieler eine Pseudozufallszahl (gleichverteilt zwischen 0 und 1) generiert; ist diese kleiner als die berechnete Wahrscheinlichkeit $p_x$, dass $x$ gewinnt, wird $x$ als Sieger bestimmt, sonst $y$. Die Anzahl der Siege jedes Spielers wird in einem Array gespeichert. Daraus wird der Turniersieger bestimmt; ist dies Lina, wird die Anzahl ihrer Siege inkrementiert.

[10] Die Reihenfolge ist von der Aufgabenstellung nicht vorgegeben, wir treffen hier aber diese naheliegende Wahl.
[11] Der Wahrscheinlichkeitswert 1 beschreibt mathematisch eine Wahrscheinlichkeit von 100%.
[12] Die Aufgabenstellung lässt offen, was in diesem Fall zu tun ist.

### Turnierform *K.O.*

Der Unterschied zur Turnierform *Liga* liegt darin, dass nicht jeder Spieler gegen jeden anderen antritt. Stattdessen werden Runden durchgeführt.

Die erste Runde wird durch einen Turnierplan $t_1, \ldots, t_n$ festgelegt, den das Programm einliest. Es treten dann Spieler $t_1$ gegen $t_2$, $t_3$ gegen $t_4$, …, und $t_{n-1}$ gegen $t_n$ an. Nachdem diese Spiele alle – genauso wie oben – simuliert wurden, werden die Gewinner in unveränderter Reihenfolge in einen neuen Turnierplan geschrieben. Dieser hat nurmehr $\frac{n}{2}$ Einträge. Dieses Verfahren wird so lange wiederholt, bis nur noch ein Spieler im Turnierplan steht; dieser ist der Turniersieger.

Ansonsten ist der Ablauf derselbe wie bei der *Liga*.

### Turnierform *K.O.*×*5*

Ein Turnier dieser Form läuft fast genauso ab wie ein K.O.-Turnier. Der einzige Unterschied ist, dass jeweils zwei Spieler nicht nur einmal gegeneinander spielen, sondern fünfmal, und von beiden der Spieler mit mehr Siegen gewinnt. Wir können also das bisherige Zweipersonenspiel zwischen Spielern $x$ und $y$ im Turnierablauf ersetzen durch fünf solche Spiele und das Problem so auf die bisherige Lösung zurückführen – wir müssen nur die Wahrscheinlichkeit bestimmen, dass Spieler $x$ bzw. Spieler $y$ unter fünf Spielen häufiger gewinnt. Wir fragen also: Wie hoch ist die Wahrscheinlichkeit, dass Spieler $x$ die Mehrzahl der Spiele gewinnt, d. h. genau 3, genau 4 oder genau 5 Spiele? Die Wahrscheinlichkeit, dass er genau 3 Spiele gewinnt, ist $\binom{5}{3} \cdot p_x^3 \cdot p_y^2$.[13] Analoge Formeln gelten für die Wahrscheinlichkeit, dass er genau 4 oder genau 5 Spiele gewinnt. Die Wahrscheinlichkeit, dass er mehr Spiele gewinnt als sein Gegner und damit die Runde gewinnt, ergibt sich so zu

$$\begin{aligned}&\binom{5}{3} \cdot p_x^3 \cdot p_y^2 + \binom{5}{4} \cdot p_x^4 \cdot p_y^1 + \binom{5}{5} \cdot p_x^5 \cdot p_y^0 \\ =&10 \cdot p_x^3 \cdot p_y^2 + 5 \cdot p_x^4 \cdot p_y + p_x^5.\end{aligned}$$

Die Wahrscheinlichkeit, dass Spieler $y$ gewinnt, ist 1 minus dieser Wert. Indem wir die Wahrscheinlichkeiten des ursprünglichen Zweipersonenspiels durch diese Wahrscheinlichkeiten ersetzen, haben wir das Problem auf die Turnierform *K.O.* zurückgeführt.

Alternativ kann man das Turnier auch tatsächlich wie beschrieben simulieren – also zwischen zwei Spielern immer pseudozufällige Ergebnisse von fünf Spielen generieren.

### Mathematische Anmerkung

Der Anteil der von Lina „im Durchschnitt über viele Wiederholungen“ gewonnenen Turniere ist gleich der Wahrscheinlichkeit, dass ein einzelnes Turnier von ihr gewonnen wird. Mathematisch lässt sich diese Gleichheit mit dem *Schwachen Gesetz der Großen Zahlen* sauber

[13] Hierbei ist $\binom{a}{b} = \frac{a!}{b!(a-b)!}$ der *Binomialkoeffizient* von $a$ über $b$. Er gibt an, wie viele Möglichkeiten es gibt, aus $a$ unterscheidbaren Objekten eine Anzahl von $b$ unter Nichtbeachtung ihrer Reihenfolge auszuwählen. $\binom{5}{3}$ ist also die Anzahl der Möglichkeiten, wie 3 Spiele von $x$ und die anderen 2 von $y$ gewonnen werden können. Die Wahrscheinlichkeit, dass 3 bestimmte Spiele von $x$ gewonnen werden, ist $p_x^3$, die Wahrscheinlichkeit, dass $y$ die anderen beiden gewinnt, $p_y^2$.

begründen.[14]

Ein Turnier im Sinne der Aufgabe ist ein sogenanntes *Zufallsexperiment*. Bezeichnet man den Gewinn eines Turniers durch Lina als *Erfolg* und den Verlust als *Misserfolg*, handelt es sich in dieser Kernfrage um ein sogenanntes *Bernoulli-Experiment*. Die Aufgabe fordert nichts anderes, als die tatsächliche Wahrscheinlichkeit $\mu$ des „Erfolgs“ in diesem Experiment zu ermitteln.

Unser Ansatz war, das Experiment $H$ mal zu simulieren und den Anteil $m$ der gemessenen „Erfolge“ als Schätzwert für $\mu$ zu betrachten.

Nach der *Tschebyscheff-Ungleichung* ist (siehe der Link von oben) die Wahrscheinlichkeit, dass der so ermittelte Schätzwert $m$ von dem tatsächlichen (theoretischen) Wert $\mu$ um mehr als 1 Prozentpunkt abweicht, kleiner als $\frac{1}{H\cdot(1\%)^2}$, wobei $H$ die Anzahl der simulierten Turniere ist. Mit $H = 10^7$ ergibt dies nur eine Wahrscheinlichkeit von $0{,}1\%$ einer Abweichung von mehr als 1 Prozentpunkt.

## 3.2 Alternativer Lösungsansatz

### Aufzählen

Eine Lösung ohne das Durchführen von Zufallsexperimenten wäre ebenfalls denkbar. Ein naheliegender Ansatz ist, alle möglichen Kombinationen von Ausgängen der einzelnen Spiele durchzuprobieren. Indem man die Wahrscheinlichkeit jeder Kombination, mit der Lina das Turnier gewinnen würde, berechnet, und diese Wahrscheinlichkeiten alle aufsummiert, bekommt man den gesuchten Wert. Dies ist, im Gegensatz zum vorigen Ansatz, tatsächlich eine exakte Formel, d. h. das Ergebnis ließe sich so – bis auf Rundungsfehler – exakt berechnen.

Dieser Ansatz hat eine Laufzeit von $\Theta(S\cdot 2^S)$, wobei $S$ die Anzahl der in der gewählten Turnierform ausgetragenen Spiele ist: Es gibt nämlich für jedes Spiel 2 mögliche Ausgänge, also $2^S$ mögliche Kombinationen von Spielausgängen, und die jeweilige Wahrscheinlichkeit und der jeweilige Turniergewinner lassen sich mit $\Theta(S)$ Schritten berechnen. Die letztere Berechnung lässt sich jedoch auch beim Aufzählen der Kombinationen nebenbei durchführen – dann erhält man eine bessere Laufzeit von $\Theta(2^S)$.

In einem Turnier der Form *Liga* werden $S = \binom{n}{2} = \frac{n(n-1)}{2}$ Spiele durchgeführt, in einem der Form *K.O.* hingegen $S = \frac{n}{2} + \frac{n}{4} + \cdots + 4 + 2 + 1 = n - 1$ (man beachte, dass $n$ hier eine Zweierpotenz sein muss) und in einem der Form *K.O.*$\times 5$, indem wir es wie oben beschrieben auf eines der Form *K.O.* zurückführen, genauso viele. Für $n = 16$ (der größte Wert aus den im Material enthaltenen Beispielen, die auf jeden Fall gelöst werden sollen) ergibt sich $S = 120$ bzw. $S = 15$. Bei einer Laufzeit von $\Theta(2^S)$ lässt sich daher abschätzen, dass dieser Ansatz für die Turnierform *Liga* nicht schnell genug wäre (aufgrund der Größenordnung von $2^{120} \approx 10^{36}$ Rechenschritten), um das Ergebnis zu liefern, für die Turnierformen *K.O.* und *K.O.*$\times 5$ jedoch schon ($2^{15} \approx 3\cdot 10^5$).

[14] Siehe dazu `https://de.wikipedia.org/wiki/Gesetz_der_grossen_Zahlen#Schwaches_Gesetz_für_relative_Häufigkeiten`

## 3.3 Beispiele

### Auswertung

Bei der Auswertung der Ergebnisse sollte beachtet werden, dass die ermittelten Wahrscheinlichkeiten nur über eine Stichprobe angenähert wurden. Daher kommt es natürlich bei wiederholter Simulation zu Schwankungen im Ergebnis – sofern der Pseudozufallszahlengenerator bei jeder Ausführung anders initialisiert wird.

Daher sollten die gemessenen Werte nicht wesentlich genauer angegeben werden, als die Genauigkeit (entspricht etwa der Schwankung in den Werten) das zulässt! Ein Beispiel: Wenn 10 Ausführungen der Simulation die folgenden Werte ausgeben:

$$34.57\%, 34.64\%, 34.68\%, 34.64\%, 34.63\%, 34.62\%, 34.57\%, 34.59\%, 34.64\%, 34.57\%,$$

dann ist es sinnvoll, das Gesamtergebnis als 34.6% zu notieren und nicht als 34.570%.

### Ergebnisse

Es wurden die folgenden unterschiedlichen Turnierpläne für die KO-Formen ausprobiert:

| Turnierplan | 8-Spieler | 16-Spieler |
|---|---|---|
| A | 1–2, 3–4, 5–6, 7–8 | 1–2, 3–4, 5–6, 7–8, 9–10, 11–12, 13–14, 15–16 |
| B | 1–8, 2–7, 3–6, 4–5 | 1–16, 2–15, 3–14, 4–13, 5–12, 6–11, 7–10, 8–9 |
| C | 1–5, 2–6, 3–7, 4–8 | 1–9, 2–10, 3–11, 4–12, 5–13, 6–14, 7–15, 8–16 |
| Zufall | Für jede Simulationsdurchführung zufällig neu bestimmt. | |

Bei einer auf- oder absteigenden Sortierung der Spieler spielen also im Turnierplan A die Spieler zunächst mit annähernd gleichstarken Gegenspielern, während im Turnierplan B die schwächsten Spieler zunächst gegen die stärksten Gegenspieler spielen. Turnierplan C versucht die Unterschiede zwischen allen Spielern im ersten Spiel hoch, aber ungefähr gleich zu machen. Sofern im ersten Spiel aber oft der stärkere Gegenspieler gewinnt, sehen in der zweiten KO-Runde aber die Pläne B und C sehr ähnlich aus. Daher wären auch noch besser randomisierte Turnierpläne denkbar.

Jede Turnierform (einschließlich der KO-Varianten) wurde $10^6$-mal simuliert. In Tabelle 1 sind die Ergebnisse aufgeführt, jeweils mit einer Genauigkeit von etwa 0.1%. In Abbildung 3.1 sind die Ergebnisse als Diagramm dargestellt.

## 3.4 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Zweipersonenspiel nicht richtig umgesetzt**
  Die im Programm benutzten Gewinnwahrscheinlichkeiten müssen korrekt aus den Spielstärken berechnet werden, und der Sieger eines Spiels muss korrekt ermittelt werden.
  Wenn zwei Spieler mit Spielstärke 0 gegeneinander antreten, funktioniert die Urnenmethode zur Ermittlung eines Gewinners nicht, da aus einer leeren Urne keine Kugel gezogen werden kann. In den BWINF-Beispielen tritt dieser Fall nicht auf. Wenn er nicht beachtet wird, führt das *nicht* zu Punktabzug.

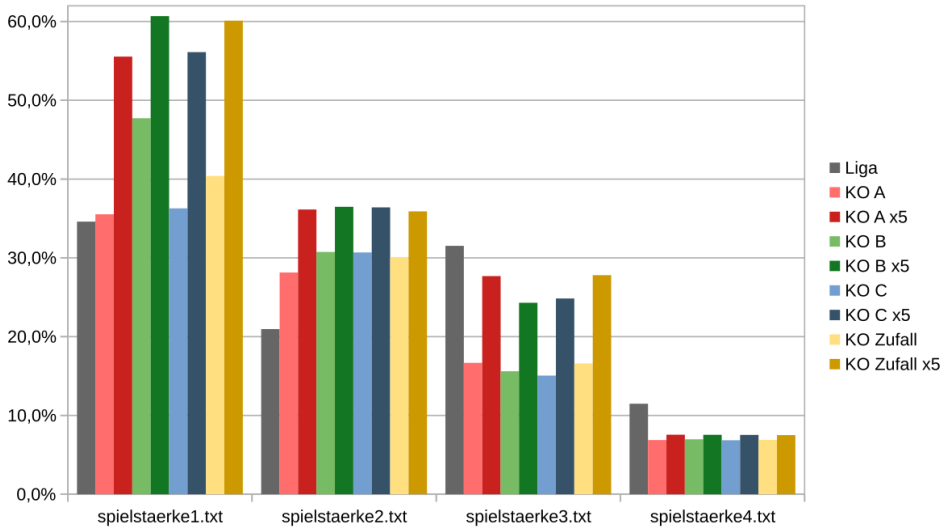



Abbildung 3.1: Ergebnisse

| | spielstaerke1.txt | spielstaerke2.txt | spielstaerke3.txt | spielstaerke4.txt |
|---|---|---|---|---|
| Liga | 34,6% | 21,0% | 31,5% | 11,5% |
| KO A | 35,5% | 28,1% | 16,7% | 6,9% |
| KO A x5 | 55,5% | 36,1% | 27,7% | 7,6% |
| KO B | 47,7% | 30,8% | 15,6% | 7,0% |
| KO B x5 | 60,7% | 36,5% | 24,3% | 7,5% |
| KO C | 36,3% | 30,7% | 15,1% | 6,9% |
| KO C x5 | 56,1% | 36,4% | 24,8% | 7,5% |
| KO Zufall | 40,4% | 30,1% | 16,6% | 6,9% |
| KO Zufall x5 | 60,1% | 35,9% | 27,8% | 7,5% |

Tabelle 1: Ergebnisse

- $[-2]$ **Turnier nicht richtig umgesetzt**
  Die Turnierabläufe müssen richtig simuliert werden. In Turnierform Liga muss im Fall eines Punktgleichstands der Spieler mit der niedrigeren Nummer gewinnen. Passieren Fehler in einer oder zwei Turnierformen, wird ein Punkt abgezogen; bei Fehlern in allen Turnierformen werden zwei Punkte abgezogen.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Das Programm muss in angemessener Zeit für alle vorgegebenen Beispiele, in allen Turnierformen, ein (approximatives) Ergebnis liefern können.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Für mindestens zwei der vorgegebenen Beispiele müssen Ergebnisse dokumentiert werden; und zwar für jede Turnierform. Dabei sind ausnahmsweise auch Übersichten außerhalb der Dokumentation akzeptabel, sofern in der Dokumentation darauf verwiesen wird; eine Darstellung der Ergebnisse in einem Tabellendokument bietet sich bei dieser Aufgabe an.
  Da sehr viele Faktoren die Ergebnisse beeinflussen können, insbesondere die Anzahl der Simulationen pro Turnierform, können die dokumentierten Ergebnisse von den hier präsentierten etwas abweichen. Deutliche Abweichungen deuten allerdings auf Fehler hin. In den Einsendungen wird bei den KO-Formen in der Regel die Variante A simuliert.
  Die Aufgabenstellung fordert eine Empfehlung an Tobi. Da diese sich im Wesentlichen aus den Ergebnissen ableitet, wird bei ihrem Fehlen auf Punktabzug verzichtet.

# Aufgabe 4: Streichholzrätsel

## 4.1 Grundidee

Es wird eine Verschiebung der Ausgangsanordnung gesucht, die möglichst viele Streichhölzer mit der Zielanordnung gemeinsam hat. Anschließend werden die nicht gemeinsamen Streichhölzer aus der Anfangsanordnung entfernt und mit ihnen die Zielanordnung konstruiert.

## 4.2 Modellierung

Zur Beschreibung einer Streichholzanordnung benötigen wir eine Möglichkeit, die einzelnen Streichhölzer darzustellen. Hier soll pro Streichholz ein Paar von zweidimensionalen Vektoren verwendet werden, welche die beiden Endpunkte des Streichholzes angeben. Eine Streichholzanordnung ist dann eine Menge solcher Vektorpaare.

### Vektorstruktur

Um nun einen einzelnen zweidimensionalen Vektor abzuspeichern, wird ein geeignetes Datenformat benötigt. Im Normalfall werden für Vektorstrukturen bzw. Vektorkoordinaten Fließkommazahlen verwendet. Dies bringt jedoch Rundungsfehler mit sich, was sich vor allem bei größeren Streichholzanordnungen bemerkbar macht.

Um dieses Problem zu umgehen, ist eine genauere Betrachtung der möglichen Vektoren notwendig. Klar ist: Jeder Vektor, dessen Winkel ein Vielfaches von 90° beträgt, kann einfach als Paar von ganzen Zahlen dargestellt werden (Abbildung 4.1). Ein Streichholz B, das am Ende ei-

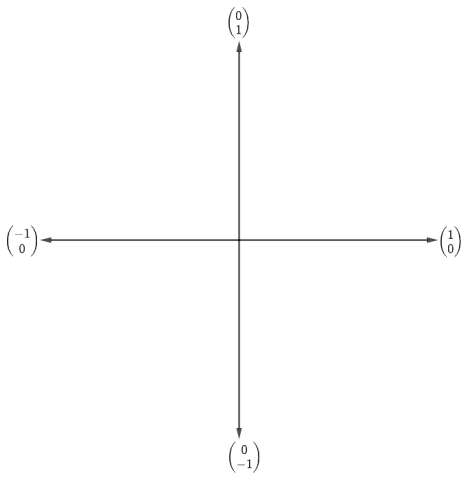



Abbildung 4.1: 90° Vektoren

nes anderen Streichholzes A anliegt, kann nun durch Addition von Vektoren dargestellt werden. Hierfür werden zu beiden Enden von B das gemeinsame Ende von A addiert (Abbildung 4.2). Bei der Betrachtung aller möglichen Additionen der 90° Vektoren kann eine bestimmte Menge von Vektoren erzeugt werden. Alle so erreichbaren Vektoren haben die Form $\begin{pmatrix} x \\ y \end{pmatrix}$ mit $x, y \in \mathbb{Z}$. Dies wird auch die Ebene der ganzen Zahlen genannt.

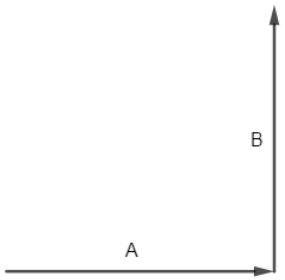



Abbildung 4.2: Addition

Als nächstes wird ein Vektor mit einem Winkel von 30° betrachtet.

$$V30 = \begin{pmatrix} \cos(30°) \\ \sin(30°) \end{pmatrix} = \frac{1}{2} \begin{pmatrix} \sqrt{3} \\ 1 \end{pmatrix}$$

Wird nun die die Länge eines Streichholzes auf 2 Einheiten gesetzt, verschwindet der Faktor $\frac{1}{2}$ vor dem Vektor und es bleibt $\begin{pmatrix} \sqrt{3} \\ 1 \end{pmatrix}$. Es ist zu beobachten, dass der Y-Wert eine ganze Zahl und der X-Wert eine irrationale Zahl ist. Dies hat zur Folge, dass der X-Wert, sowohl durch Addition als auch Multiplikation (ausgenommen 0) mit ganzen Zahlen, nicht mehr rational werden kann. Daraus folgt, dass, solange der Faktor vor $\sqrt{3}$ nicht wieder 0 wird, keiner der Vektoren in der Ebene der ganzen Zahlen erreichen werden kann.

Wird ein Winkel von 60° gewählt, kann die gleiche Aussage über das Verhalten der Y-Koordinate getroffen werden.

$$V60 = \begin{pmatrix} \cos(60°) \\ \sin(60°) \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 \\ \sqrt{3} \end{pmatrix}$$

Da alle noch übrigen Winkel durch Spiegelung an der X- oder Y-Achse erzeugt werden können, hat jeder Vektor die Form

$$\begin{pmatrix} x + z \cdot \sqrt{3} \\ y + w \cdot \sqrt{3} \end{pmatrix}$$

mit $x, y, z, w \in \mathbb{Z}$. Um die weitere Darstellung zu erleichtern, wird jeder Wert in einer eigenen Zeile des Vektors gesetzt.

$$\begin{pmatrix} x + z \cdot \sqrt{3} \\ y + w \cdot \sqrt{3} \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

So entstehen folgende Vektoren für alle 12 Winkel:

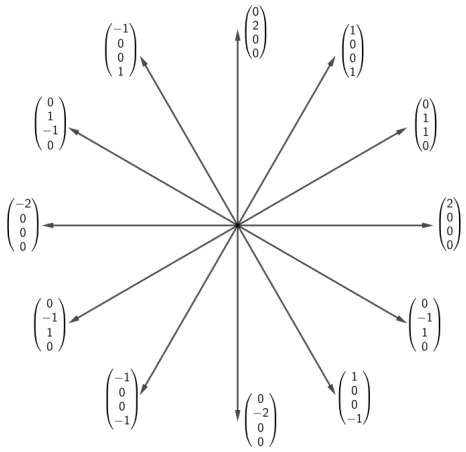


Abbildung 4.3: Vektoren

Die Addition solcher Vektoren ist nun wie folgt definiert:

$$\begin{pmatrix} x_0 + z_0 \cdot \sqrt{3} \\ y_0 + w_0 \cdot \sqrt{3} \end{pmatrix} + \begin{pmatrix} x_1 + z_1 \cdot \sqrt{3} \\ y_1 + w_1 \cdot \sqrt{3} \end{pmatrix} = \begin{pmatrix} x_0 + x_1 + (z_0 + z_1) \cdot \sqrt{3} \\ y_0 + y_1 + (w_0 + w_1) \cdot \sqrt{3} \end{pmatrix}$$

Daraus folgt:

$$\begin{pmatrix} x_0 \\ y_0 \\ z_0 \\ w_0 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ w_1 \end{pmatrix} = \begin{pmatrix} x_0 + x_1 \\ y_0 + y_1 \\ z_0 + z_1 \\ w_0 + w_1 \end{pmatrix}$$

Da der Algorithmus nicht nur Translation, sondern auch Rotation betrachtet, wird noch eine Formel benötigt, die einen beliebigen Vektor um 30° rotiert. Hierfür wird die Rotationsmatrix für die Ebene verwendet.

$$\begin{pmatrix} \cos(30°) & -\sin(30°) \\ \sin(30°) & \cos(30°) \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix} \tag{4.1}$$

$$\begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix} \begin{pmatrix} x + z \cdot \sqrt{3} \\ y + w \cdot \sqrt{3} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} (x + z \cdot \sqrt{3})\sqrt{3} - (y + w \cdot \sqrt{3}) \\ x + z \cdot \sqrt{3} + (y + w \cdot \sqrt{3})\sqrt{3} \end{pmatrix} \tag{4.2}$$

$$= \frac{1}{2} \begin{pmatrix} 3z - y + (x - w) \cdot \sqrt{3} \\ x + 3w + (y + z) \cdot \sqrt{3} \end{pmatrix} \tag{4.3}$$

Nach einer Rotation um 30° entsteht also folgender Vektor:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \mapsto \frac{1}{2} \begin{pmatrix} 3z - y \\ x + 3w \\ x - w \\ y + z \end{pmatrix}$$

**Algorithmus 4** Lösung

```
 1: procedure VERGLEICHE(m1,m2)                          ▷ Anordnungen m1, m2
 2:     for all i ∈ {1,...,12} do
 3:         for all Winkel w do                           ▷ 0° bis 150°
 4:             for all Streichhölzer s_i in m1[w] do     ▷ (*)
 5:                 for all Streichhölzer s_j in m2[w] do
 6:                     Verschiebe m1 so, dass s_i auf s_j liegt.
 7:                     Zähle überdeckende Streichhölzer c.
 8:                     if N − K = c then
 9:                         Gib restliche Streichhölzer zurück.
10:                     end if
11:         Rotiere m2 um 30°.                            ▷ (**)
12: end procedure
```

(*)Wobei $m[w]$ alle Streichhölzer aus Anordnung $m$ sind, die Winkel $w$ haben.
(**)Bei der Rotation einer Anordnung werden für alle Vektorpaare beide Vektoren 30° um den Koordinatenursprung rotiert.

#### Vergleichs-Algorithmus

Mit diesen Vektoren ist es nun möglich, die Streichholzanordnungen darzustellen. Im nächsten Schritt werden zwei Streichholzanordnungen miteinander verglichen: Es wird nach einer Verschiebung gesucht, welche die korrekte Anzahl an Streichhölzern in beiden Anordnungen übereinstimmen lässt. Die Anzahl der Streichhölzer heißt im weiteren Verlauf $N$, und die Anzahl umzulegender Streichhölzer heißt $K$.

Der Algorithmus sortiert zunächst die Streichhölzer einer Anordnung. Dafür werden sie in 6 Mengen unterteilt. Die 6 Mengen stehen für die 6 Winkel von 0° bis 150°. Jedes Streichholz, dass in keine dieser Mengen einsortiert werden kann, wird um 180° gedreht (es werden Anfangs- und Endpunkt getauscht) und anschließend einsortiert.

Nachdem dies mit beiden Anordnungen geschehen ist, läuft das Verfahren wie in Algorithmus 4 beschrieben ab. Es werden also je zwei Streichhölzer übereinander gelegt und anschließend wird ausgezählt, wie viele Streichhölzer bei dieser Verschiebung übereinander liegen. Nachdem alle Verschiebungen ausprobiert wurden, wird die Zielanordnung um 30° rotiert und der Algorithmus beginnt von vorn.

## 4.3 Laufzeitbestimmung

Für eine Laufzeitbestimmung werden die einzelnen Schleifenlängen betrachtet. Für die erste Schleife in Zeile 2 ist dieser Wert 12, für die 12 möglichen Winkel, also $\mathcal{O}(12) = \mathcal{O}(1)$. In der zweiten, dritten und vierten Schleife (Zeile 3 bis 5) wird über alle Streichholzpaare iteriert, also $\mathcal{O}(N^2)$. Das anschließende Zählen der überdeckenden Streichhölzer dauert $\mathcal{O}(N)$. Hierbei wird für alle $N$ Streichhölzer aus der Ausgangsanordnung getestet, ob sie mit einem Streichholz aus der Zielanordnung übereinstimmen. Letzteres lässt sich in konstanter Zeit umsetzen, wenn für die Streichhölzer Mengen verwendet werden, die konstante Suchzeit für ihre Elemente haben. So entsteht eine Gesamtlaufzeit T mit

$$T(N) = \mathcal{O}(N^3)$$

.

## 4.4 Beispiele

Die Beispiele der BWINF-Website müssen zunächst in Textdateien übersetzt werden. Eine solche Textdatei enthält in ihrer ersten Zeile die Anzahl $N$ der Streichhölzer und in der darauf folgenden Zeile die Anzahl $K$ der umzulegenden Streichhölzer. Die folgenden Zeilen beschreiben jeweils ein Streichholz und enthalten dazu den Start- und Endvektor. Nach den Zeilen für die Ausgangsanordnung wird eine Zeile frei gelassen, um die Zeilen für die Zielanordnung auch optisch abzusetzen. Für das auch in der Aufgabenstellung abgebildete Beispiel (streichhoelzer0.png) entsteht die Textdatei aus Abbildung 4.4.

```
7
3
0 0 0 0|2 0 0 0
0 0 0 0|0 2 0 0
2 0 0 0|2 2 0 0
0 2 0 0|2 2 0 0
0 2 0 0|1 2 0 1
2 2 0 0|1 2 0 1
1 2 0 1|1 4 0 1

0 0 0 0|0 2 0 0
0 0 0 0|1 0 0 1
1 0 0 1|2 0 0 0
2 0 0 0|2 2 0 0
1 0 0 1|1 2 0 1
0 2 0 0|1 2 0 1
2 2 0 0|1 2 0 1
```

Abbildung 4.4: Textdatei

Das Programm sucht nach allen Textdateien, die im selben Ordner wie die Anwendung liegen, und gibt ihre Lösung in einer gleichnamigen Bilddatei aus und als Text in der Konsole. Die entstandenen Bilder haben einen Farbcode. Eine schwarze Linie steht für ein Streichholz, das liegen bleibt, eine rote Linie für ein Streichholz, das entfernt werden muss, und eine grüne Linie für ein Streichholz, das dort hingelegt werden muss, um die Zielanordnung zu erreichen. Die Lösungen für die sechs Beispieldateien von der BWINF-Website werden in den Abbildungen 4.5 bis 4.10 gezeigt.

Da keines dieser Beispiele davon profitiert, dass auch Rotationen berücksichtigt werden, muss hierzu ein extra Beispiel generiert werden, siehe Abbildung 4.11.

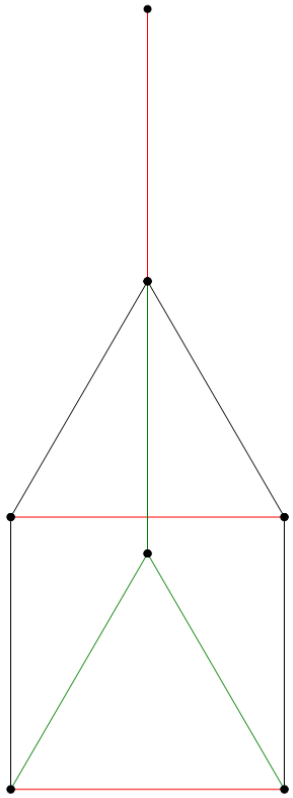


Abbildung 4.5: Lösung für streichhoelzer0.png

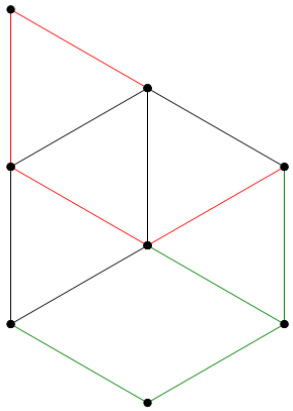


Abbildung 4.6: Lösung für streichhoelzer1.png

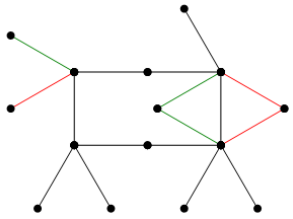


Abbildung 4.7: Lösung für streichhoelzer2.png

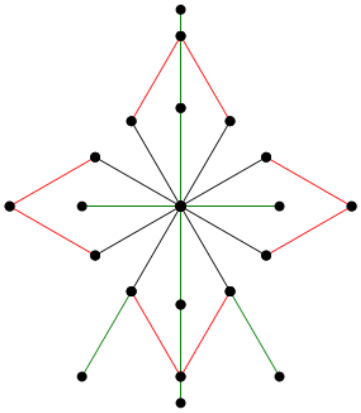


Abbildung 4.8: Lösung für streichhoelzer3.png

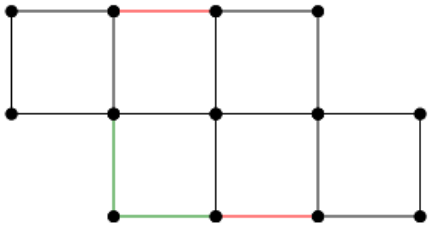


Abbildung 4.9: Lösung für streichhoelzer4.png

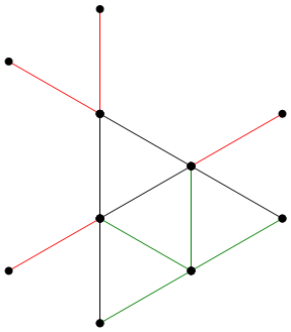


Abbildung 4.10: Lösung für streichhoelzer5.png

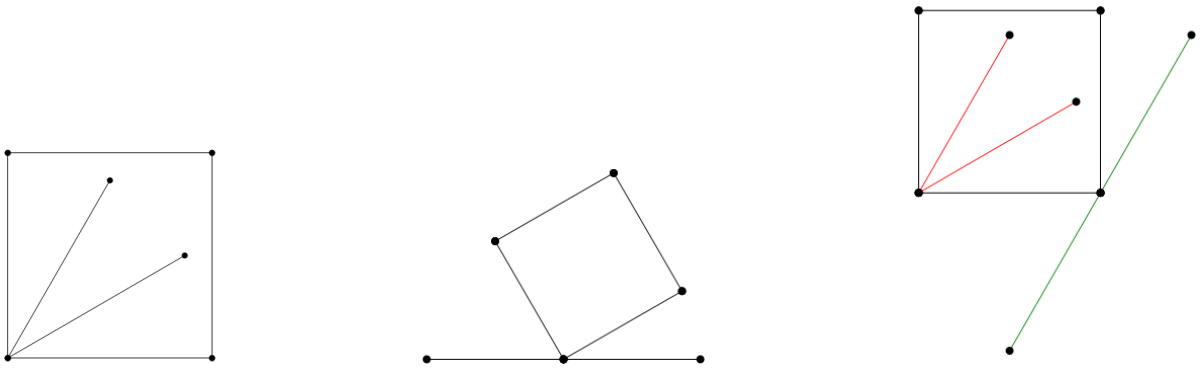


Abbildung 4.11: Weiteres Beispiel: Ausgangsanordnung, Zielanordnung und Lösung

## 4.5 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Modellierung ungeeignet**
  In der Aufgabenstellung wird ein „Format zur Darstellung im Rechner“ gefordert. Damit ist eine interne Repräsentation gemeint; ein vom Programm lesbares Eingabeformat schadet nichts, ist aber nicht gefordert. Die interne Darstellung diese die Geometrie der Anordnungen berücksichtigen. Sie kann, wie gezeigt, auf einem diskreten Gitter und den 12 diskreten Richtungen basieren. Die Verwendung von Fließkommazahlen ist also nicht nötig. Wenn Fließkommazahlen verwendet werden, muss mit möglichen Rundungsfehlern geeignet umgegangen werden – siehe nächstes Kriterium.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das Lösungsverfahren muss stets eine Lösung finden, wenn es eine gibt. Es ist allerdings in Ordnung (und mit der Aufgabenstellung in Einklang), wenn zu einem Rätsel, bei dem die Zielanordnung insgesamt rotiert ist, keine Lösung erkannt wird. Wenn bei Verwendung von Fließkommazahlen Rundungsfehler zu Problemen führen, wird ein Punkt abgezogen.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Der Vergleich zweier Anordnungen sollte durch das beschriebene Verfahren in angemessener Zeit möglich sein. Laufzeiten, die deutlich über $O(n^3)$ liegen, insbesondere also bei nicht-polynomieller Laufzeit, führen zu Punktabzug.
- $[-1]$ **Ausgabe schlecht nachvollziehbar**
  Die zu verschiebenden Streichhölzer müssen erkennbar markiert sein. Die Ausgabe muss nicht zwingend grafisch erfolgen. Die Anordnung der Streichhölzer und die zur Lösung nötigen Verschiebungen müssen aber erkennbar sein. Daher ist eine ASCII-Art-Ausgabe oder Ähnliches sinnvoll. Es ist aber auch akzeptabel, wenn die Ausgabe des Programms sozusagen „von Hand“ in nachvollziehbare Abbildungen umgesetzt wird – die aber in der Dokumentation enthalten sein müssen, siehe nächstes Kriterium. Leider sind in vielen Einsendungen keine visuell nachvollziehbaren Darstellungen vorhanden.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Es sollen Ergebnisse zu mindestens drei der fünf zusätzlichen Beispiele zu der Aufgabe (`streichhoelzer1.txt` bis `streichhoelzer5.txt`) in der Dokumentation abgedruckt sein.

# Aufgabe 5: Wichteln

## 5.1 Lösungsidee

In dieser Aufgabe soll eine Wichtel-Verteilung gefunden werden, die zuerst möglichst viele Erstwünsche, dann möglichst viele Zweitwünsche und schließlich möglichst viele Drittwünsche erfüllt.

Wir nutzen hier die „Ungarischen Methode“[15], um die Aufgabe zu lösen. Dafür mussen wir das Problem zunächst ein wenig umformulieren, so dass wir es als Optimierungsproblem beschreiben können. Wir wollen die Erfüllung von Wünschen so mit Punkten belohnen, dass ein Zuteilung der Geschenke mit maximaler Punktzahl auch optimal im Sinne der Aufgabenstellung ist.

Sei $n \geq 3$ die Anzahl der Geschenke bzw. Personen. Eine Verteilung der Geschenke an die Personen wird Matching genannt. Jedes Matching erhält abhängig von der Art und Anzahl der erfüllten Wünsche eine Punktzahl. Beispielsweise können wir $n^2+n+1$ Punkte für jeden erfüllten Erstwunsch, $n+1$ Punkte für jeden erfüllten Zweitwunsch und 1 Punkt für jeden erfüllten Drittwunsch vergeben. Die konkrete Menge an Punkten, die jede Wunschart wert ist, kann auch anders gewählt werden. Eine hinreichende Bedingung ist, dass ein Erstwunsch strikt mehr Punkte wert ist als maximal viele Zweitwünsche $(n^2+n+1 > n \cdot (n+1))$, dass ein Zweitwunsch strikt mehr Punkte wert ist als maximal viele Drittwünsche $(n+1 > n \cdot 1)$ und dass ein Drittwunsch strikt mehr als 0 Punkte wert ist $(1 > 0)$.

Die Gewichtung der Wunscharten stellt sicher, dass ein Matching mit maximaler Punktzahl im Sinne der Aufgabenstellung optimal ist. Mit der Ungarischen Methode wird dann ein solches Matching bestimmt.

### Darstellung

Sei $G = \{g_1, \ldots, g_n\}$ die Menge der Geschenke und $P = \{p_1, \ldots, p_n\}$ die Menge der Personen. Betrachtet wird das Beispiel in Tabelle 2.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| Erstwunsch | $g_2$ | $g_2$ | $g_4$ | $g_4$ | $g_5$ |
| Zweitwunsch | $g_1$ | $g_3$ | $g_2$ | $g_2$ | $g_3$ |
| Drittwunsch | $g_4$ | $g_4$ | $g_5$ | $g_1$ | $g_1$ |

Tabelle 2: Beispiel für $n = 5$.

Es wird eine $n \times n$ große Punktetabelle angelegt. Jede Zeile steht für ein Geschenk und jede Spalte für eine Person. Eine Zelle beschreibt, wie viele Punkte die Vergabe des Geschenks an die Person wert ist. Tabelle 3 ist die Punktetabelle für das Beispiel in Tabelle 2.

Die Ungarische Methode liefert letztlich ein Matching mit maximaler Punktzahl.

[15] https://de.wikipedia.org/wiki/Ungarische_Methode

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | 6 | 0 | 0 | 1 | 1 |
| $g_2$ | 31 | 31 | 6 | 6 | 0 |
| $g_3$ | 0 | 6 | 0 | 0 | 6 |
| $g_4$ | 1 | 1 | 31 | 31 | 0 |
| $g_5$ | 0 | 0 | 1 | 0 | 31 |

Tabelle 3: Erstwünsche sind $n^2+n+1=31$ Punkte wert, Zweitwünsche sind $n+1=6$ Punkte wert, Drittwünsche sind 1 Punkt wert und alles andere ist 0 Punkte wert.

## 5.2 Ungarische Methode

Werden in einer Tabelle $n$ Zellen so markiert, dass in jeder Zeile und in jeder Spalte genau eine markierte Zelle liegt, dann beschreiben die Markierungen ein Matching: Ein Geschenk $g$ wird genau dann an eine Person $p$ verteilt, wenn die Zelle in der Spalte $p$ und in der Zeile $g$ markiert ist. Die Punktzahl dieses Matchings ist die Summe der markierten Zellen. Ein Beispiel für ein Matching in Tabelle 3 zeigt Tabelle 4.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | 6 | 0 | 0 | **1** | 1 |
| $g_2$ | **31** | 31 | 6 | 6 | 0 |
| $g_3$ | 0 | **6** | 0 | 0 | 6 |
| $g_4$ | 1 | 1 | **31** | 31 | 0 |
| $g_5$ | 0 | 0 | 1 | 0 | **31** |

Tabelle 4: Ein Tupel $(p,g)$ beschreibt, dass Person $p$ Geschenk $g$ erhält. Die grau markierten Zellen beschreiben das Matching $\{(p_1,g_2),(p_2,g_3),(p_3,g_4),(p_4,g_1),(p_5,g_5)\}$ mit der Punktzahl $31+6+31+1+31=100$.

### Umformung in ein nicht-negatives Minimierungsproblem

Ein Matching heißt minimal, wenn es eine minimale Punktzahl hat. Tabelle 5 wird konstruiert, indem jede Zelle in Tabelle 3 mit $-1$ multipliziert wird. Jedes maximale Matching in Tabelle 3 ist dann ein minimales Matching in Tabelle 5. Gesucht werden jetzt minimale Matchings.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | -6 | 0 | 0 | -1 | -1 |
| $g_2$ | -31 | -31 | -6 | -6 | 0 |
| $g_3$ | 0 | -6 | 0 | 0 | -6 |
| $g_4$ | -1 | -1 | -31 | -31 | 0 |
| $g_5$ | 0 | 0 | -1 | 0 | -31 |

Tabelle 5: Alle Zellen in Tabelle 3 werden mit $-1$ multipliziert.

Die Werte in der Tabelle sind jetzt aber alle negativ.

Jetzt machen wir uns die folgende Einsicht zu Nutze: Wenn in einer Tabelle eine Zeile oder Spalte mit einer Konstante $c \in \mathbb{Z}$ addiert wird, dann steigt die Punktzahl jedes Matchings um

genau $c$, weil in jedem Matching genau eine Zelle pro Zeile und Spalte markiert ist. Ein minimales Matching bleibt also auch nach einer solchen Umformung weiterhin minimal. Diese Idee wird im weiteren Verlauf wiederholt verwendet, um Tabelle 5 so umzuformen, dass das Finden eines minimalen Matchings möglichst leicht fällt.

Tabelle 6 wird konstruiert, indem alle Spalten in Tabelle 5 mit $n^2+n+1$ addiert werden. Außerdem sind dann alle Werte in Tabelle 6 nicht-negativ, weil $-n^2-n-1$ der betragsgrößte negative Wert in Tabelle 5 ist.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | 25 | 31 | 31 | 30 | 30 |
| $g_2$ | 0 | 0 | 25 | 25 | 31 |
| $g_3$ | 31 | 25 | 31 | 31 | 25 |
| $g_4$ | 30 | 30 | 0 | 0 | 31 |
| $g_5$ | 31 | 31 | 30 | 31 | 0 |

Tabelle 6: Zu allen Zellen in Tabelle 5 wird $n^2+n+1=31$ addiert.

### Subtraktion der Zeilenminima

Tabelle 7 wird konstruiert, indem in Tabelle 6 das Minimum jeder Zeile bestimmt und von der Zeile subtrahiert wird. Diese Umformung stellt sicher, dass die Werte in Tabelle 7 nicht-negativ bleiben und in jeder Zeile mindestens eine 0 steht.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ | min |
|---|---|---|---|---|---|---|
| $g_1$ | 25 | 31 | 31 | 30 | 30 | 25 |
| $g_2$ | 0 | 0 | 25 | 25 | 31 | 0 |
| $g_3$ | 31 | 25 | 31 | 31 | 25 | 25 |
| $g_4$ | 30 | 30 | 0 | 0 | 31 | 0 |
| $g_5$ | 31 | 31 | 30 | 31 | 0 | 0 |

$\Longrightarrow$

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | 0 | 6 | 6 | 5 | 5 |
| $g_2$ | 0 | 0 | 25 | 25 | 31 |
| $g_3$ | 6 | 0 | 6 | 6 | 0 |
| $g_4$ | 30 | 30 | 0 | 0 | 31 |
| $g_5$ | 31 | 31 | 30 | 31 | 0 |

Tabelle 7: Tabelle 6 mit subtrahierten Zeilenminima.

### Subtraktion der Spaltenminima

Tabelle 8 wird konstruiert, indem in Tabelle 7 das Minimum jeder Spalte bestimmt und von der Spalte subtrahiert wird. Diese Umformung stellt sicher, dass die Werte in Tabelle 8 nicht-negativ bleiben und in jeder Spalte mindestens eine 0 steht.

### Ausführung der Strichmethode

Jedes Matching in Tabelle 8 hat eine nicht-negative Punktzahl. Außerdem liegt in jeder Zeile und Spalte mindestens eine 0. Es ist nicht garantiert, dass ein Matching mit der Punktzahl 0 existiert, aber wenn ein Matching die Punktzahl 0 hat, dann ist es minimal. Die folgende Methode, die Strichmethode genannt wird, bestimmt entweder, ob ein Matching mit der Punktzahl 0 existiert, oder verteilt die Nullen durch geschickte Umformungen. Die Strichmethode wird solange hintereinander ausgeführt, bis ein optimales Matching gefunden wird.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | 0 | 6 | 6 | 5 | 5 |
| $g_2$ | 0 | 0 | 25 | 25 | 31 |
| $g_3$ | 6 | 0 | 6 | 6 | 0 |
| $g_4$ | 30 | 30 | 0 | 0 | 31 |
| $g_5$ | 31 | 31 | 30 | 31 | 0 |
| min | 0 | 0 | 0 | 0 | 0 |

$\Rightarrow$

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | 0 | 6 | 6 | 5 | 5 |
| $g_2$ | 0 | 0 | 25 | 25 | 31 |
| $g_3$ | 6 | 0 | 6 | 6 | 0 |
| $g_4$ | 30 | 30 | 0 | 0 | 31 |
| $g_5$ | 31 | 31 | 30 | 31 | 0 |

Tabelle 8: Tabelle 7 mit subtrahierten Spaltenminima.

Ein Strich erstreckt sich über eine ganze Zeile oder Spalte. Es wird eine minimale Anzahl an Strichen gesucht, die alle Nullen in der Tabelle streichen. Tabelle 9 zeigt eine minimale Anzahl von Strichen in Tabelle 8, die alle Nullen streicht.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | 0 | 6 | 6 | 5 | 5 |
| $g_2$ | 0 | 0 | 25 | 25 | 31 |
| $g_3$ | 6 | 0 | 6 | 6 | 0 |
| $g_4$ | 30 | 30 | 0 | 0 | 31 |
| $g_5$ | 31 | 31 | 30 | 31 | 0 |

Tabelle 9: Vier Striche erstrecken sich über die Spalten $p_1, p_2, p_5$ und über die Zeile $g_4$. Es werden strikt weniger als $n = 5$ Striche benötigt, um alle Nullen zu streichen. Das Minimum der nicht gestrichenen Zellen ist $h = 5$.

Wenn $n$ Striche benötigt werden, um alle Nullen zu streichen, dann existieren $n$ Nullen, die alle in verschiedenen Zeilen und Spalten liegen. Also existiert dann ein Matching mit der Punktzahl 0. Wenn strikt weniger als $n$ Striche benötigt werden, um alle Nullen zu streichen, dann wird das Minimum $h$ der nicht gestrichenen Zellen bestimmt. Jede nicht gestrichene Spalte wird mit $h$ subtrahiert und jede gestrichene Zeile wird mit $h$ addiert. Tabelle 10 zeigt das Ergebnis dieser Umformung von Tabelle 9.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | 0 | 6 | 1 | 0 | 5 |
| $g_2$ | 0 | 0 | 20 | 20 | 31 |
| $g_3$ | 6 | 0 | 1 | 1 | 0 |
| $g_4$ | 35 | 35 | 0 | 0 | 36 |
| $g_5$ | 31 | 31 | 25 | 26 | 0 |

Tabelle 10: Es werden die nicht gestrichenen Spalten $p_3, p_4$ mit $h = 5$ subtrahiert und die gestrichene Zeile $g_4$ mit $h = 5$ addiert. In anderen Worten werden die nicht gestrichenen grau markierten Zellen mit $h = 5$ subtrahiert und die doppelt gestrichenen schwarz markierten Zellen mit $h = 5$ addiert.

Diese Umformung garantiert, dass die Werte in der Tabelle weiterhin nicht-negativ sind. Außerdem steigt die Anzahl der Striche, die benötigt werden, um alle Nullen abzudecken. Wenn die Methode genügend oft hintereinander ausgeführt wird, dann erreicht die Anzahl der benötigten Striche den Wert $n$.

Um alle Nullen in Tabelle 10 mit Strichen abzudecken, werden jetzt $n = 5$ Striche benötigt. Also existiert ein Matching mit der Punktzahl 0 in Tabelle 10. Tabelle 11 zeigt ein Matching mit der Punktzahl 0 in Tabelle 10.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $g_1$ | 0 | 6 | 1 | 0 | 5 |
| $g_2$ | 0 | 0 | 20 | 20 | 31 |
| $g_3$ | 6 | 0 | 1 | 1 | 0 |
| $g_4$ | 35 | 35 | 0 | 0 | 36 |
| $g_5$ | 31 | 31 | 25 | 26 | 0 |

Tabelle 11: Das gleiche Matching wie in Tabelle 4 ist eine optimale Lösung.

## 5.3 Optimierung

Wenn eine Person $p$ einen einzigartigen Erstwunsch $g$ hat, dann wird in jedem optimalen Matching Geschenk $g$ an Person $p$ verteilt. Tabelle 12 zeigt alle einzigartigen Erstwünsche in Tabelle 2. Das Problem wird auf die Größe $m \leq n$ reduziert, indem die einzigartigen Erstwünsche vor Ausführung der Ungarischen Methode vergeben werden. Tabelle 13 ist diese Reduktion von Tabelle 2. Tabelle 14 zeigt die Punktetabelle für Tabelle 13.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| Erstwunsch | $g_2$ | $g_2$ | $g_4$ | $g_4$ | $g_5$ |
| Zweitwunsch | $g_1$ | $g_3$ | $g_2$ | $g_2$ | $g_3$ |
| Drittwunsch | $g_4$ | $g_4$ | $g_5$ | $g_1$ | $g_1$ |

Tabelle 12: Person $p_5$ hat den einzigartigen grau markierten Erstwunsch $g_5$.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|
| Erstwunsch | $g_2$ | $g_2$ | $g_4$ | $g_4$ |
| Zweitwunsch | $g_1$ | $g_3$ | $g_2$ | $g_2$ |
| Drittwunsch | $g_4$ | $g_4$ | | $g_1$ |

Tabelle 13: In Tabelle 12 wird die Spalte der zufriedengestellten Person $p_5$ und die Zellen des verteilten Geschenks $g_5$ gelöscht. Übrig bleibt ein Problem der Größe $m = 4$.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|---|
| $g_1$ | 5 | 0 | 0 | 1 |
| $g_2$ | 21 | 21 | 5 | 5 |
| $g_3$ | 0 | 5 | 0 | 0 |
| $g_4$ | 1 | 1 | 21 | 21 |

Tabelle 14: Erstwünsche sind $m^2 + m + 1 = 21$ Punkte wert, Zweitwünsche sind $m + 1 = 5$ Punkte wert, Drittwünsche sind 1 Punkt wert und alles andere ist 0 Punkte wert. Die Spalte $p_5$ und die Zeile $g_5$ existieren nicht mehr.

| Beispiel | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Erstwünsche | 6 | 3 | 15 | 15 | 13 | 37 | 541 |
| Zweitwünsche | 0 | 0 | 6 | 4 | 1 | 3 | 127 |
| Drittwünsche | 2 | 0 | 1 | 3 | 7 | 21 | 58 |

Tabelle 15: Anzahl der erfüllten Wünsche in einem optimalen Matching

## 5.4 Beispiele

Tabelle 15 dient zur Überprüfung der eigenen Ergebnisse zu den Beispieleingaben von der BWINF-Website.

Für einige kleinere Beispiele werden nun konkrete Verteilungen der Geschenke angegeben.

**wichteln1.txt**

```
Kind 1 erhält Geschenk 6 (3. Wunsch)
Kind 2 erhält Geschenk 2 (1. Wunsch)
Kind 3 erhält Geschenk 1 (3. Wunsch)
Kind 4 erhält Geschenk 3 (1. Wunsch)
Kind 5 erhält Geschenk 5
Kind 6 erhält Geschenk 4 (1. Wunsch)
Kind 7 erhält Geschenk 7 (einzigartiger 1. Wunsch)
Kind 8 erhält Geschenk 10 (einzigartiger 1. Wunsch)
Kind 9 erhält Geschenk 9 (einzigartiger 1. Wunsch)
Kind 10 erhält Geschenk 8
```

**wichteln2.txt**

```
Kind 1 erhält Geschenk 1
Kind 2 erhält Geschenk 2
Kind 3 erhält Geschenk 3
Kind 4 erhält Geschenk 6 (1. Wunsch)
Kind 5 erhält Geschenk 5 (1. Wunsch)
Kind 6 erhält Geschenk 4 (1. Wunsch)
Kind 7 erhält Geschenk 7
Kind 8 erhält Geschenk 8
Kind 9 erhält Geschenk 9
Kind 10 erhält Geschenk 10
```

**wichteln3.txt**

```
Kind 1 erhält Geschenk 2 (einzigartiger 1. Wunsch)
Kind 2 erhält Geschenk 1
Kind 3 erhält Geschenk 29 (1. Wunsch)
Kind 4 erhält Geschenk 8 (1. Wunsch)
Kind 5 erhält Geschenk 20 (1. Wunsch)
Kind 6 erhält Geschenk 3 (1. Wunsch)
Kind 7 erhält Geschenk 5
Kind 8 erhält Geschenk 12 (einzigartiger 11. Wunsch)
Kind 9 erhält Geschenk 4 (1. Wunsch)
Kind 10 erhält Geschenk 28 (einzigartiger 11. Wunsch)
Kind 11 erhält Geschenk 6 (1. Wunsch)
Kind 12 erhält Geschenk 9 (1. Wunsch)
Kind 13 erhält Geschenk 14 (2. Wunsch)
Kind 14 erhält Geschenk 23 (2. Wunsch)
Kind 15 erhält Geschenk 11
Kind 16 erhält Geschenk 13
Kind 17 erhält Geschenk 18
Kind 18 erhält Geschenk 7 (2. Wunsch)
Kind 19 erhält Geschenk 16 (einzigartiger 11. Wunsch)
Kind 20 erhält Geschenk 10 (einzigartiger 11. Wunsch)
Kind 21 erhält Geschenk 19 (2. Wunsch)
Kind 22 erhält Geschenk 22
Kind 23 erhält Geschenk 24
Kind 24 erhält Geschenk 15 (2. Wunsch)
Kind 25 erhält Geschenk 17 (3. Wunsch)
Kind 26 erhält Geschenk 26 (1. Wunsch)
Kind 27 erhält Geschenk 27 (1. Wunsch)
Kind 28 erhält Geschenk 21 (2. Wunsch)
Kind 29 erhält Geschenk 25
Kind 30 erhält Geschenk 30 (1. Wunsch)
```

**wichteln4.txt**

```
Kind 1 erhält Geschenk 23
Kind 2 erhält Geschenk 21 (einzigartiger 1. Wunsch)
Kind 3 erhält Geschenk 4
Kind 4 erhält Geschenk 26
Kind 5 erhält Geschenk 6 (2. Wunsch)
Kind 6 erhält Geschenk 5 (3. Wunsch)
Kind 7 erhält Geschenk 3 (2. Wunsch)
Kind 8 erhält Geschenk 16 (3. Wunsch)
Kind 9 erhält Geschenk 9
Kind 10 erhält Geschenk 19 (1. Wunsch)
Kind 11 erhält Geschenk 10
Kind 12 erhält Geschenk 11
Kind 13 erhält Geschenk 7 (3. Wunsch)
```

```
Kind 14 erhält Geschenk 14 (1. Wunsch)
Kind 15 erhält Geschenk 2 (einzigartiger 1. Wunsch)
Kind 16 erhält Geschenk 20
Kind 17 erhält Geschenk 1 (einzigartiger 1. Wunsch)
Kind 18 erhält Geschenk 27 (1. Wunsch)
Kind 19 erhält Geschenk 17 (2. Wunsch)
Kind 20 erhält Geschenk 13 (einzigartiger 1. Wunsch)
Kind 21 erhält Geschenk 22 (2. Wunsch)
Kind 22 erhält Geschenk 28 (1. Wunsch)
Kind 23 erhält Geschenk 15 (einzigartiger 1. Wunsch)
Kind 24 erhält Geschenk 12 (1. Wunsch)
Kind 25 erhält Geschenk 30 (einzigartiger 1. Wunsch)
Kind 26 erhält Geschenk 25
Kind 27 erhält Geschenk 18 (einzigartiger 1. Wunsch)
Kind 28 erhält Geschenk 8 (einzigartiger 1. Wunsch)
Kind 29 erhält Geschenk 24 (1. Wunsch)
Kind 30 erhält Geschenk 29 (einzigartiger 1. Wunsch)
```

## 5.5 Weiterführende Literatur

Diese Aufgabe ist in der Literatur als ein Rank-maximal Matchingproblem bekannt. In diesem Lösungshinweis wurde die Aufgabe auf ein gewichtetes Matchingproblem zurückgeführt und mit der Ungarischen Methode gelöst, was bei einer optimalen Implementierung eine Laufzeit in $\mathcal{O}(n^3)$ hat. Naive Implementierungen, wie zum Beispiel eine einfache Breiten- oder Tiefensuche, haben nur eine exponentielle Laufzeit. Robert Irving et al. haben einen Algorithmus entdeckt, der diese Aufgabe mit einer Laufzeit in $\mathcal{O}(n^2)$ löst.

1. Wikipedia, Ungarische Methode
   `https://de.wikipedia.org/wiki/Ungarische_Methode`
2. Wikipedia, Rank-maximal Matching (engl.)
   `https://en.wikipedia.org/wiki/Rank-maximal_allocation`
3. Robert Irving et. al., Rank-Maximal Matchings:
   `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.6742&rep=rep1&type=pdf`

## 5.6 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Lösungsverfahren fehlerhaft**
  Die gefundenen Geschenke-Verteilungen müssen grundsätzlich korrekt sein: Geschenke dürfen nicht doppelt vergeben werden, Personen dürfen nicht leer ausgehen.
- $[-1]$ **Gefundene Verteilungen nicht (annähernd) optimal**
  Im allgemeinen handelt es sich hier um ein Matching-Problem. Ein Lösungsverfahren kann auf der ungarischen Methode basieren oder ähnlich funktionieren. Alternativ kann das Problem aber auch mit Fluss-Algorithmen gelöst werden. Auch gute Heuristiken sind

akzeptabel. Sie sollten einzigartige Erstwünsche berücksichtigen und auch im weiteren die Vergabe eher seltener Erstwünsche priorisieren. Mit guten Heuristiken lassen sich die Beispiele 1 bis 4 oder sogar 5 in der Regel optimal lösen; bei Beispiel 6 kann die Anzahl der vergebenen Drittwünsche leicht abweichen; bei Beispiel 7 sollte die Anzahl der vergebenen Erstwünsche optimal sein und die Anzahl der vergebenen Zweitwünsche nicht allzu deutlich vom Optimum abweichen.

- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Das Verfahren sollte für alle vorgegebenen Beispiel in angemessener Zeit eine Verteilung finden. Nicht-polynomielle Laufzeiten sind nicht akzeptabel.
- $[-1]$ **Ausgabe schlecht nachvollziehbar**
  Aus der Ausgabe sollte ersichtlich werden, welche Person welches Geschenk erhält und welchem Wunsch es entspricht. Letzteres ist in der Aufgabenstellung zwar nicht gefordert, aber zur (auch eigenen) Einschätzung des Ergebnisses offensichtlich von entscheidender Bedeutung.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Es sollten mindestens drei der sieben Beispiele zu der Aufgabe (`wichteln1.txt` bis `wichteln7.txt`) in der Dokumentation abgedruckt sein.