37. Bundeswettbewerb Informatik 2018/2019, 1. Runde

Lösungshinweise und Bewertungskriterien

Bundeswettbewerb Informatik

Allgemeines

Es war sehr erfreulich, dass sich wieder besonders viele die Zeit zur Bearbeitung der Aufgaben nahmen. Die BewerterInnen gaben sich ebenfalls Mühe und 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. Einige unterhaltsame Formulierungsperlen der Informatik sind im Anhang wiedergegeben.

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 erhielt man eine Bewertung der Einsendung, die bei der Anfertigung künftiger Lösungen hilfreich sein kann.

Die Bearbeitungszeit für die 1. Runde beträgt etwa drei Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist daher der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der wichtigen, ausführlichen Dokumentation der Aufgabenlösungen zu vermeiden. Einige Aufgaben sind oft schwerer zu lösen, 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. Daher ist es sinnvoll, einzureichende Aufgaben nicht nacheinander, sondern relativ gleichzeitig zu bearbeiten, um nicht vom zeitlichen Aufwand einer Aufgabe unangenehm kurz vor Einsendeschluss überrascht zu werden und keine vollständige Einreichung mehr zu schaffen.

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

Dokumentation

Die Zeit für die Bewertung ist leider begrenzt, weshalb es unmöglich ist, alle eingesandten Programme gründlich zu testen. Die wichtige Grundlage der Bewertung ist deshalb stets die Dokumentation, die, wie im Aufgabenblatt beschrieben, für jede bearbeitete Aufgabe aus den Teilen *Lösungsidee*, *Umsetzung*, *Beispielen* und *Quellcode* besteht. Leider sind die Dokumentationen bei vielen Einsendungen sehr knapp ausgefallen, was oft zu Punktabzügen führte, die das Erreichen der 2. Runde verhinderten.

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. Grundsätzlich kann die Dokumentation einer Aufgabe als „sehr unverständlich bzw. unvollständig“ bewertet werden, wenn die meisten Inhalte kaum verständlich sind oder Teile wie Lösungsidee, Umsetzung oder Quellcode komplett fehlen.

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 Eingabendaten/-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 des BwInf 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), genug 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 in der Dokumentation erwartet werden. Zu jeder Aufgabe werden deshalb im jeweils letzten Abschnitt die Bewertungskriterien näher erläutert, worauf bei der Bewertung dieser Aufgabe besonders geachtet wird. Die Kriterien sind in der Bewertung, die man im PMS einsehen kann, aufgelistet und geben an, inwieweit eine Einsendung die einzelnen Bewertungskriterien jeweils bezüglich einer Aufgabe erfüllt. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation, die oben erklärt sind.

In der 1. Runde des *Bundeswettbewerbs Informatik (BwInf)* 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 bei einem Kriterium kein Punktabzug ist, ist die Einsendung in Bezug auf dieses Kriterium in Ordnung. Wenn das Kriterium dagegen nicht erfüllt ist, gibt es einen Punkt Abzug. Wurde die Aufgabe nur unzureichend bearbeitet, wird ein gesondertes Kriterium angewandt, bei dem es vier oder fünf Punkte Abzug gibt. Im schlechtesten Fall wird eine Aufgabenbearbeitung also mit 0 Punkten bewertet.

Für die Gesamtpunktzahl des BwInf 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 des BwInf 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 BwInf-Einsendung nicht beachtet, vgl. 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 BwInf gewertet werden.

Danksagung

Alle Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt: Peter Rossmanith, *RWTH Aachen University* (Vorsitzender); Hanno Baehr, *RWE Supply & Trading GmbH, Essen*; Jens Gallenbacher, *TU Darmstadt, JGU Mainz*; Rainer Gemulla, *Universität Mannheim*; Torben Hagerup, *Universität Augsburg*; Christof Hanke, *Berufliches Gymnasium für Informatik, FLB Herford*; Thomas Kesselheim, *Universität Bonn*; Arno Pasternak, *Fritz-Steinhoff-Gesamtschule Hagen, TU Dortmund*; Holger Schlingloff, *Fraunhofer FOKUS, Berlin*; Melanie Schmidt, *Universität Bonn*; als Gäste im Ausschuss: Mario Albrecht und Wolfgang Pohl, *BWINF, Bonn*.

An der Erstellung der im folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Tamás Korodi (Junioraufgabe 1), Thomas Leineweber (Junioraufgabe 2), Michael Rosenthal (Aufgabe 1), Maximilian Azendorf (Aufgabe 2), Tamás Korodi (Aufgabe 3), Dominik Meier (Aufgabe 4) und Julian Baldus (Aufgabe 5). Allen Beteiligten sei für Ihre Mitarbeit hiermit ganz herzlich gedankt.

Junioraufgabe 1: Auf und Ab

J1.1 Lösungsidee

Wir beschreiben jede Leiter des Leiterspiels durch ein Zahlenpaar (a, b). Die erste Zahl a gibt die Feldnummer an, bei welchem die Leiter anfängt, und die zweite Zahl b die Feldnummer, bei welchem die Leiter endet.

Beispiel

Das Paar $(6, 27)$ entspricht einer Leiter, die beim Feld mit der Nummer 6 anfängt und beim Feld mit der Nummer 27 aufhört.

Das ganze Spielbrett lässt sich so durch eine Liste von Leitern darstellen (vgl. Abbildung J1.1):

(6, 27), (14, 19), (21, 53), (31, 42), (33, 38), (46, 62),
(51, 59), (57, 96), (65, 85), (68, 80), (70, 76), (92, 98)

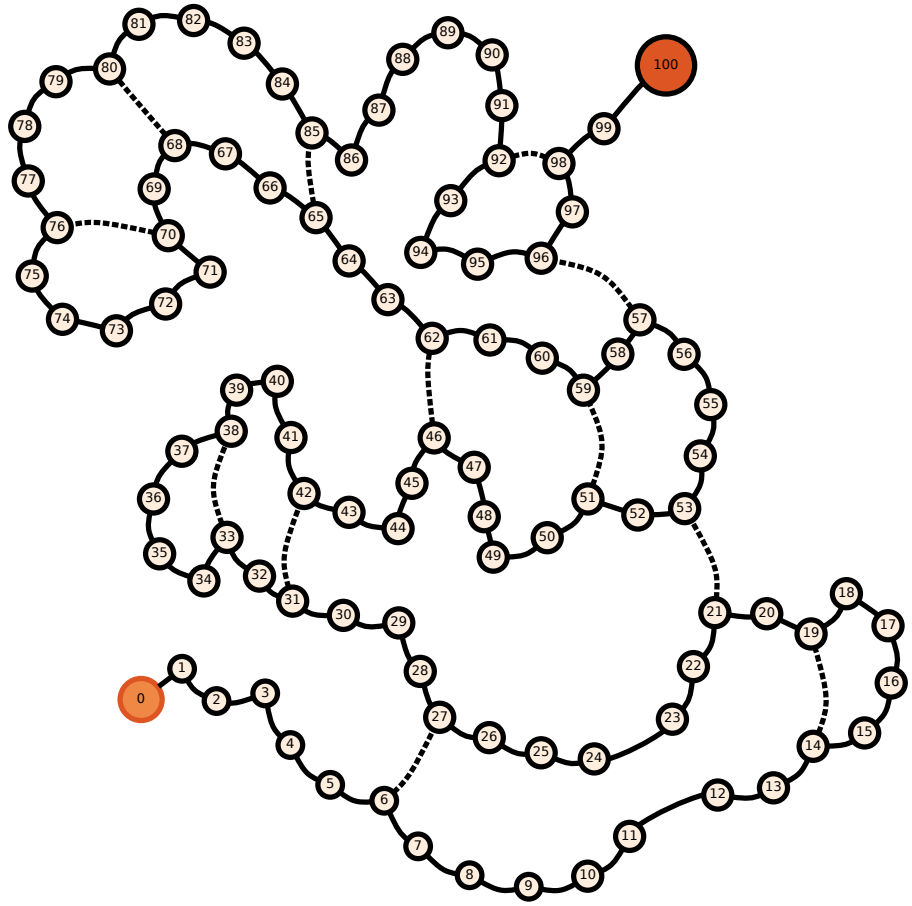


Abbildung J1.1: Übersicht über das Spielbrett. Die gestrichelten Linien repräsentieren die Leitern.

J1.2 Berechnung eines Spielzugs

Wir speichern die jeweilige Feldnummer, auf der sich Fredericks Spielfigur befindet, in einer Variable P. Nach Würfeln einer Zahl Z werden zur Bestimmung der neuen Figurenposition folgende Schritte ausgeführt:

1. Erhöhe die Figurenposition P um die gewürfelte Zahl Z.
2. Überprüfe, ob die neue Figurenposition P nun über 100 liegt. In diesem Fall sind wir über das Ziel hinausgelaufen und müssen die überzähligen Schritte wieder zurückgehen. Dazu ersetzen wir P durch $100 - (P - 100)$, also $200 - P$.
3. Überprüfe, ob an der neuen Figurenposition P eine Leiter beginnt oder endet. Hierzu suchen wir in der Liste der Zahlenpaare nach der Position P. Gibt es ein Paar mit der Zahl P, so befindet sich Fredericks Spielfigur an einer Leiter; die neue Position P wird dann durch die andere Nummer dieses Zahlenpaars bestimmt (Auf-/Abstieg durch eine Leiter).

Es ist wichtig, dass wir die Berechnungen in genau dieser Reihenfolge durchführen. Es kann nämlich passieren, dass die Spielfigur erst über das Ziel hinausläuft und beim Zurücklaufen (Zielrücküberquerung) dann auch noch auf eine Leiter trifft, an der sie hinabsteigen muss, z.B. falls bei Feldnummer 99 eine Drei gewürfelt wird und somit von Feld 98 zu Feld 92 hinuntergeklettert werden muss.

J1.3 Erkennen von Endlosschleifen

Die Aufgabenstellung ist, ob wir durch das Würfeln derselben Zahl ins Ziel kommen. Dazu setzen wir unsere Spielfigur anfangs auf das Startfeld 0. Wir berechnen nacheinander die einzelnen Spielzüge mit der vorher genannten Methode. Dabei zählen wir nebenbei die Anzahl der notwendigen Züge mit. Sind wir im Zielfeld 100 angekommen, so können wir mit der Berechnung aufhören (vgl. Abbildung J1.3).

Allerdings tritt noch ein Problem auf. Fredericks Spielfigur kann sich auch in einer Endlosschleife verfangen und wird deswegen das Zielfeld nie erreichen (vgl. Abbildung J1.2). Daher müssen wir sicher testen können, ob wir in einer solchen Endlosschleife gefangen sind. Dazu gibt es verschiedene Möglichkeiten.

Möglichkeit 1: Wir merken uns die Nummer der Felder, auf denen Fredericks Spielfigur am Ende eines Zuges schon war. Kommt seine Spielfigur nach einem Spielzug auf ein bereits besuchtes Feld, so weiß er, dass er ab jetzt immer im Kreis laufen wird (wir nehmen ja an, dass wir immer die gleiche Zahl würfeln).

Möglichkeit 2: Das gesamte Spielbrett besteht aus 100 verschiedenen Feldern (Startfeld nicht mitgezählt). Wenn sich Fredericks Spielfigur nicht in einer Endlosschleife verfangen hat, so muss seine Spielfigur schon nach höchstens 100 Zügen im Zielfeld angekommen sein. Andernfalls hätte er nämlich eine bestimmte Nummer mehrfach besucht, und sich so in einer Endlosschleife verfangen. Wir können die Berechnung also spätestens nach 100 Spielzügen abbrechen.

Möglichkeit 3: Frederick sucht sich eine Mitspielerin Franziska. Beide würfeln immer die gleiche Zahl Z. Wenn Frederick einen Zug macht, macht Franziska währenddessen zweimal diesen Zug. Sie ist also doppelt so schnell unterwegs wie Frederick. Treffen sich beide Spieler auf irgendeinem Spielfeld zwischen 1 und 99, so muss es eine Endlosschleife geben. Wenn Franziska

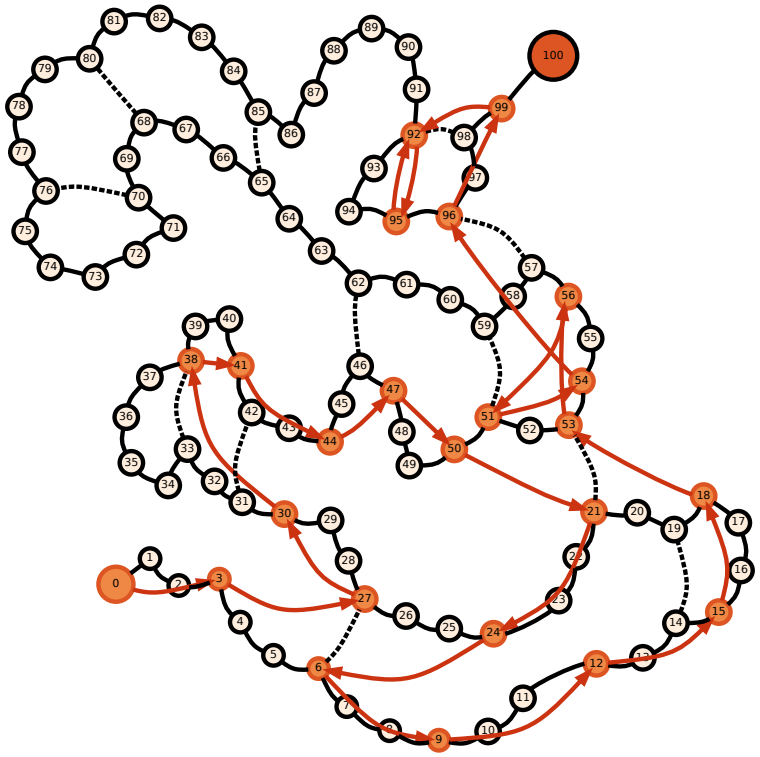


Abbildung J1.2: Für die Würfelzahl Drei landet Frederick in einer Endlosschleife.

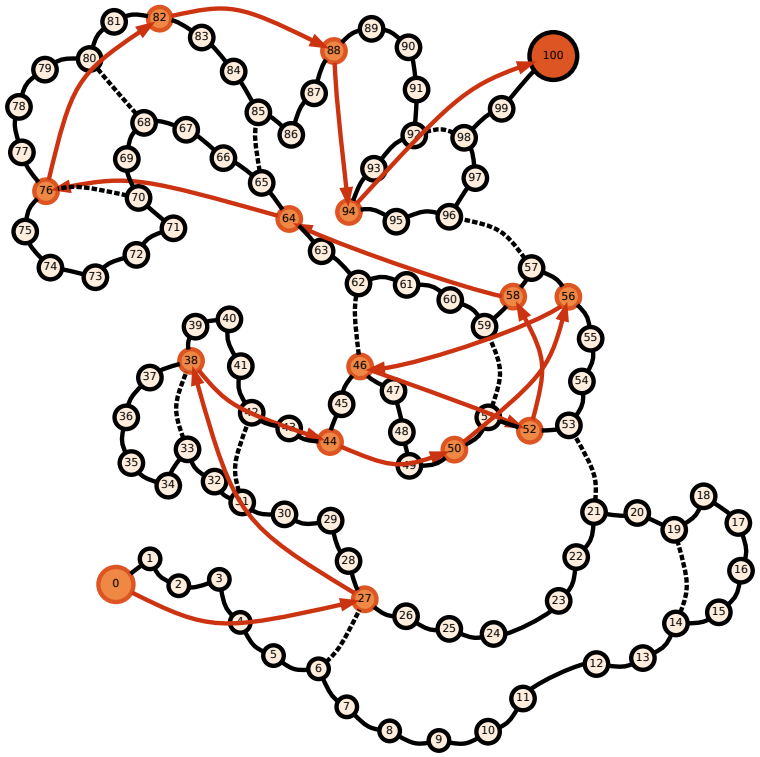


Abbildung J1.3: Für die Würfelzahl Sechs kommt Frederick im Ziel an.

nämlich doppelt so schnell im Kreis läuft wie Frederick, so wird sie ihn irgendwann von hinten einholen. Gibt es keine Endlosschleife, so treffen sich Franziska und Frederick erst wieder im Zielfeld 100, aber auf keinen Fall davor. Dieses Verfahren ist auch unter dem Namen *Hase-Igel-Algorithmus* bekannt und hat den Vorteil, dass man sich weder den Weg noch die Anzahl der Schritte merken muss.

Möglichkeit 4: Das Programm bricht nach einer Minute Rechenzeit als Zeitlimit automatisch ab, da aufgrund der relativ langen Programmlaufzeit bei korrekter Programmierung nur eine Endlosschleife für Fredericks Spielfigur in Frage kommt.

J1.4 Beispiele

Im Folgenden werden die jeweiligen Ergebnisse für alle mögliche Würfelzahlen Eins bis Sechs berechnet. Während des Spielverlaufs werden die Felder ausgegeben, auf denen Frederick am Ende des jeweiligen Zuges steht. Außerdem gibt das Programm auch an, welche Leiter Frederick benutzte oder ob er zurücklaufen musste, weil er zu weit über das Ziel hinausgeschossen war. Zahlen in runden Klammern geben die relevanten Zwischenschritte während eines Zuges an.

```
1
2  Frederick würfelt jetzt immer die Zahl 1.
3  Frederick kann das Ziel erreichen.
4  Die Anzahl seiner Züge beträgt: 26
5  Frederick läuft folgenden Pfad:
6   [Start] 1 2 3 4 5 (6) [Leiter aufwärts] 27 28 29 30 (31)
7   [Leiter aufwärts] 42 43 44 45 (46) [Leiter aufwärts] 62 63 64 (65)
8   [Leiter aufwärts] 85 86 87 88 89 90 91 (92) [Leiter aufwärts] 98 99 100
9   [Im Ziel]
10
11 Frederick würfelt jetzt immer die Zahl 2.
12 Frederick kann das Ziel erreichen.
13 Die Anzahl seiner Züge beträgt: 17
14 Frederick läuft folgenden Pfad:
15  [Start] 2 4 (6) [Leiter aufwärts] 27 29 (31) [Leiter aufwärts] 42 44
16  (46) [Leiter aufwärts] 62 64 66 (68) [Leiter aufwärts] 80 82 84 86 88
17  90 (92) [Leiter aufwärts] 98 100 [Im Ziel]
18
19 Frederick würfelt jetzt immer die Zahl 3.
20 Frederick verfängt sich in einer Endlosschleife.
21 Frederick läuft folgenden Pfad:
22  [Start] 3 (6) [Leiter aufwärts] 27 30 (33) [Leiter aufwärts] 38 41 44
23  47 50 (53) [Leiter abwärts] 21 24 (27) [Leiter abwärts] 6 9 12 15 18
24  (21) [Leiter aufwärts] 53 56 (59) [Leiter abwärts] 51 54 (57)
25  [Leiter aufwärts] 96 99 (102) [zu weit] (98) [Leiter abwärts] 92
26  [Anfang der Endlosschleife] 95 (98) [Leiter abwärts] 92
27  [Ende der Endlosschleife]
28
29 Frederick würfelt jetzt immer die Zahl 4.
30 Frederick verfängt sich in einer Endlosschleife.
31 Frederick läuft folgenden Pfad:
32  [Start] 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 (68)
```

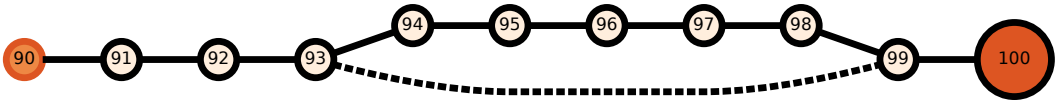
```
 [Leiter aufwärts] 80 84 88 (92) [Leiter aufwärts] 98 (102) [zu weit]
 (98) [Leiter abwärts] 92 (96) [Leiter abwärts] 57 61 (65)
 [Leiter aufwärts] 85 89 93 97
 [Anfang der Endlosschleife] (101) [zu weit] 99 (103) [zu weit] 97
 [Ende der Endlosschleife]

Frederick würfelt jetzt immer die Zahl 5.
Frederick kann das Ziel erreichen.
Die Anzahl seiner Züge beträgt: 16
Frederick läuft folgenden Pfad:
 [Start] 5 10 15 20 25 30 35 40 45 50 55 60 (65) [Leiter aufwärts] 85 90
 95 100 [Im Ziel]

Frederick würfelt jetzt immer die Zahl 6.
Frederick kann das Ziel erreichen.
Die Anzahl seiner Züge beträgt: 14
Frederick läuft folgenden Pfad:
 [Start] (6) [Leiter aufwärts] 27 (33) [Leiter aufwärts] 38 44 50 56
 (62) [Leiter abwärts] 46 52 58 64 (70) [Leiter aufwärts] 76 82 88 94
 100 [Im Ziel]
```

J1.5 Andere Leiteranordnungen

Was passiert bei anderen Leiteranordnungen? Wenn Frederick immer die Zahl Eins würfelt, kommt er immer ins Ziel, egal wie die Leitern angeordnet sind (solange zu jedem Feld von 1 bis 99 höchstens eine Leiter gehört). Für eine Endlosschleife muss es nämlich ein Feld geben, welches von zwei verschiedenen Vorgängerfeldern aus erreicht werden kann (je nachdem, ob Frederick aus der Endlosschleife kommt oder von einem Feld von außerhalb der Schleife). Nur über ein solches Feld kann er überhaupt in eine Endlosschleife geraten. Bei Leitern kann dieser Effekt nicht auftreten, denn kommt er von einem Ende der Leiter, landet er danach immer am gegenüberliegenden Ende. Allerdings kann es zwei Vorgänger geben, wenn er über das Ziel hinausschießt und dabei auf ein Feld kommt, auf dem er schon vorher war. Wenn wir immer nur die Zahl Eins würfeln, passiert so etwas aber nicht, aber für größere Würfelzahlen können Endlosschleifen auftreten.

Beispiel

Nehmen wir die folgende Konstellation:



Würfeln wir vom Feld 90 ausgehend immer die Zahl Drei, so läuft die Spielfigur wie folgt:

Mit der Feldnummer 93 beginnt eine Endlosschleife. Dementsprechend hat das Feld 93 auch die zwei Vorgängerfelder 96 und 98.

J1.6 Bewertungskriterien

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

- (1) [-1] **Dokumentation sehr unverständlich bzw. unvollständig**.
- (2) [-1] **Interne Darstellung des Spielbretts ungeeignet**:
  Für das Spielbrett mit den Leitern wurde eine geeignete interne Darstellung gefunden, d.h. die interne Darstellung ist korrekt und entspricht dem vorgegebenen Spielbrett. Auch eine feste Verdrahtung des vorgegebenen Spielfelds im Programm ist hierfür in Ordnung.
- (3) [-1] **Lösungsverfahren fehlerhaft**:
  Das Lösungsverfahren, mit dem das Würfeln von stets derselben Zahl nachvollzogen wird, sollte korrekt sein und Endlosschleifen erkannt werden. Es genügt, wenn das Verfahren für das vorgegebene Spielfeld funktioniert.
- (4) [-1] **Spielregeln mangelhaft umgesetzt**:
  Alle Spielregeln sollten korrekt umgesetzt sein (Auf-/Abstieg durch Leitern sowie Zielrücküberquerung, ggf. inklusive Leiterabstieg). Abweichungen davon sollten begründet sein.
- (5) [-1] **Verfahren bzw. Implementierung unnötig aufwendig**:
  Nur besonders umständliche Verfahren bzw. Implementierungen führen zu Punktabzug.
- (6) [-1] **Programmausgabe des Spielverlaufs nicht nachvollziehbar**:
  Die Ausgabe des Spielverlaufs durch das Programm muss nachvollziehbar sein. Eine reine Ja/Nein-Antwort je Würfelzahl ist zu wenig, jedoch reicht eine Angabe der Anzahl der Spielzüge aus. Am besten ist, wenn das Programm den gesamten Weg ins Zielfeld (oder in eine Endlosschleife) ausgibt.
- (7) [-1] **Ergebnisse fehlerhaft bzw. zu wenige (mind. 1/4 und 1/2)**:
  Die Dokumentation enthält die Ergebnisse für mind. 1 Würfelzahl aus Eins, Zwei, Drei und Sechs sowie für mind. 1 Würfelzahl aus Drei und Vier (wegen Endlosschleife).

Junioraufgabe 2: Baywatch

J2.1 Lösungsidee

Bei dieser Aufgabe muss die Piratin Longstock die beiden Landzungen-Listen passend miteinander vergleichen, um eine vollständige Landzungen-Liste zu erstellen, die mit der Landzunge im Norden beginnt. Stellen wir uns vor, sie hat die lückenhafte Liste

```
? Wald ? ? ? ? Wald See ? Wiese ? ? ?
```

und die Liste

```
Wald Wald Wiese Häuser Wüste Wald See Wald Wiese Sumpf Wüste See Häuser
```

von George. Sie kann sich schon einmal sicher sein, dass in der unvollständigen Liste nur die Fragezeichen durch konkrete Landzungen aus der zweiten Liste ersetzt werden müssen. Die zweite Liste muss an den Positionen ohne Fragezeichen genau die Landzungen der ersten Liste enthalten. Hier passt an der ersten Position zum Fragezeichen der ersten Liste der *Wald* der zweiten Liste. An der zweiten Position steht in beiden Listen ein *Wald*. Erst an der siebten Position gibt es ein Problem, weil in der ersten Liste *Wald* und in der zweiten Liste *See* steht. Damit muss die zweite Liste um eine Landzunge verschoben mit der ersten Liste verglichen werden. Wenn es dann passt, kann das Ergebnis ausgegeben werden. Wenn aber nicht, muss immer weiter verschoben werden, bis alle Möglichkeiten ausprobiert wurden.

Hier wird jetzt die zweite Liste so verschoben, dass das letzte Element der zweiten Liste entfernt und an den Anfang der zweiten Liste angefügt wird. Damit haben wir eine neue zweite Liste, in der die Elemente um eine Position verschoben sind:

```
Häuser Wald Wald Wiese Häuser Wüste Wald See Wald Wiese Sumpf Wüste See
```

Wenn diese neue zweite Liste mit der ersten Liste verglichen wird, erkennt man, dass jetzt an allen Positionen die Landzungen übereinstimmen.

Jedoch muss man die zweite Liste nicht immer wieder neu umbauen, um die beiden Listen jedes Mal miteinander vergleichen zu können. Stattdessen kann mit einer „Verschiebungszahl“ gearbeitet werden, die angibt, um wie viele Positionen sich die Landzungen in den beiden Listen unterscheiden, die im aktuellen Durchlauf zu vergleichen sind. Es wird mit der Verschiebungszahl 0 gestartet, die sich bis zu der um Eins verringerten Länge der Liste schrittweise erhöht.

```
KartenListe = Liste von der Landkarte mit Fragezeichen
GeorgListe = vollständige Liste
laenge = Länge der Kartenliste

// Alle möglichen Verschiebungen i probieren
für alle i von 0 bis (laenge-1):

  nochKorrekt = true
  für alle j von 0 bis (laenge-1):
    indexGeorg = (i + j) modulo laenge
    indexKarte = j
    wenn GeorgListe[indexGeorg] != KartenListe[indexKarte]
      und KartenListe[indexKarte] != Fragezeichen
```

```
14      dann
15        // Diese Verschiebung kann nicht mehr gültig sein
16        nochKorrekt = false
17
18    wenn nochKorrekt
19      // Verschiebung i ist möglich
20      gebe die GeorgListe mit Verschiebung i aus
21      beende das Programm
```

In den vorgegebenen Beispieldaten werden statt Wörtern nur Zahlen verwendet, die die Landzungen repräsentieren. Deshalb wird im Programm auch nur mit Zahlen statt mit Wörtern gearbeitet.

J2.2 Beispiele

Für jedes Beispiel ist im Folgenden nur jeweils eine Lösung als Ergebnis angegeben, was laut Aufgabenstellung genügt (1: Wald, 2: Wiese, 3: Häuser, 4: Wüste, 5: See, 6: Sumpf, 7: Reisfeld, 8: Berg, 9: Vulkankrater). Die jeweilige Verschiebung besagt, um wie viele Positionen die Liste von Georg nach links verschoben werden muss.

baywatch1.txt

```
1 1 2 3 4 1 5 1 2 6 4 5 3
? 1 ? ? ? ? 1 5 ? 2 ? ? ?
```

Dies ist das Beispiel vom Aufgabenblatt. Es gibt genau eine Lösung als Ergebnis mit der Verschiebung 12:

```
3 1 1 2 3 4 1 5 1 2 6 4 5
```

baywatch2.txt

```
3 2 8 2 2 8 8 1 4 1 3 7 6 7 1 5 4 5 6 3 4 1 4 2 5 6 9 9 1 5 3 2 1 3 1 6 2 8
3 9 6 4 7 1 1 9 9 5 7 5 6 5 6 7 8 5 1 2 1 8 8 7 1 7 1 7 4 9 8 7 3 8 7 1 9 8
6 1 7 6 9 8 6 1 1 1 9 1 2 7 1 3 1 8 3 5 7 9 3 1 6 5 7 4 1 8 8 3 9 2 6 2 1 5
2 2 6 3 1 6 7 1 5 6 5 4 6 8 4 8 6 5 7 4 1 9 2 4 9 6 9 9 1 1 6 1 9 3 3 9 6 1
1 8 7 7 7 7 6 7 9 4 3 7 7 8 9 5 1 9 5 4 8 6 2 5 7 5 3 4 3 5 7 5 1 5 6 2 1 8
3 4 6 1 6 1 9 5 2 1
? ? ? ? ? ? ? ? ? ? 7 ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? 5 ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? 3 ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? 8 8 ?
? 7 ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? 7 ? ? ? ? ?
? ? ? ? ? ? ? ? ? ?
```

Auch hier gibt es nur eine Lösung. Dieses Ergebnis hat die Verschiebung 148:

```
3 9 6 1 1 8 7 7 7 7 6 7 9 4 3 7 7 8 9 5 1 9 5 4 8 6 2 5 7 5 3 4 3 5 7 5 1 5
6 2 1 8 3 4 6 1 6 1 9 5 2 1 3 2 8 2 2 8 8 1 4 1 3 7 6 7 1 5 4 5 6 3 4 1 4 2
5 6 9 9 1 5 3 2 1 3 1 6 2 8 3 9 6 4 7 1 1 9 9 5 7 5 6 5 6 7 8 5 1 2 1 8 8 7
1 7 1 7 4 9 8 7 3 8 7 1 9 8 6 1 7 6 9 8 6 1 1 1 9 1 2 7 1 3 1 8 3 5 7 9 3 1
```

6 5 7 4 1 8 8 3 9 2 6 2 1 5 2 2 6 3 1 6 7 1 5 6 5 4 6 8 4 8 6 5 7 4 1 9 2 4
9 6 9 9 1 1 6 1 9 3

baywatch3.txt

2 1 3 2 8 2 2 8 8 1 4 1 3 7 6 7 1 5 4 5 6 3 4 1 4 2 5 6 9 9 1 5 3 2 1 3 1 6
2 8 3 9 6 4 7 1 1 9 9 5 7 5 6 5 6 7 8 5 1 2 1 8 8 7 1 7 1 7 4 9 8 7 3 8 7 1
9 8 6 1 7 6 9 8 6 1 1 1 9 1 2 7 1 3 1 8 3 5 7 9 3 1 6 5 7 4 1 8 8 3 9 2 6 2
1 5 2 2 6 3 1 6 7 1 5 6 5 4 6 8 4 8 6 5 7 4 1 9 2 4 9 6 9 9 1 1 6 1 9 3 3 9
6 1 1 8 7 7 7 7 6 7 9 4 3 7 7 8 9 5 1 9 5 4 8 6 2 5 7 5 3 4 3 5 7 5 1 5 6 2
1 8 3 4 6 1 6 1 9 5
? ?
? ?
? ?
? ?
? ?
? ? ? ? ? ? ? ? ? ?
Da die Landkarte nur 200 Fragezeichen enthält, funktioniert jede Verschiebung und es gibt 200 mögliche Lösungen, hier das Ergebnis mit der Verschiebung 0:
2 1 3 2 8 2 2 8 8 1 4 1 3 7 6 7 1 5 4 5 6 3 4 1 4 2 5 6 9 9 1 5 3 2 1 3 1 6
2 8 3 9 6 4 7 1 1 9 9 5 7 5 6 5 6 7 8 5 1 2 1 8 8 7 1 7 1 7 4 9 8 7 3 8 7 1
9 8 6 1 7 6 9 8 6 1 1 1 9 1 2 7 1 3 1 8 3 5 7 9 3 1 6 5 7 4 1 8 8 3 9 2 6 2
1 5 2 2 6 3 1 6 7 1 5 6 5 4 6 8 4 8 6 5 7 4 1 9 2 4 9 6 9 9 1 1 6 1 9 3 3 9
6 1 1 8 7 7 7 7 6 7 9 4 3 7 7 8 9 5 1 9 5 4 8 6 2 5 7 5 3 4 3 5 7 5 1 5 6 2
1 8 3 4 6 1 6 1 9 5

baywatch4.txt

2 1 3 2 8 2 2 8 8 1 4 1 3 7 6 7 1 5 4 5 6 3 4 1 4 2 5 6 9 9 1 5 3 2 1 3 1 6
2 8 3 9 6 4 7 1 1 9 9 5 7 5 6 5 6 7 8 5 1 2 1 8 8 7 1 7 1 7 4 9 8 7 3 8 7 1
9 8 6 1 7 6 9 8 6 1 1 1 9 1 2 7 1 3 1 8 3 5 7 9 3 1 6 5 7 4 1 8 8 3 9 2 6 2
1 5 2 2 6 3 1 6 7 1 5 6 5 4 6 8 4 8 6 5 7 4 1 9 2 4 9 6 9 9 1 1 6 1 9 3 3 9
6 1 1 8 7 7 7 7 6 7 9 4 3 7 7 8 9 5 1 9 5 4 8 6 2 5 7 5 3 4 3 5 7 5 1 5 6 2
1 8 3 4 6 1 6 1 9 5
? 5 ? ? ? ? ? ? ? ? ? ?
? ?
? 8 ? ? ? ? ? ? ?
? ?
? ?
? ? ? ? ? ? ? ? ? ?
Auch dieses Beispiel besteht aus 200 Landzungen. Es hat aber nur 2 Lösungen, hier das Ergebnis mit der Verschiebung 150:
3 9 6 1 1 8 7 7 7 7 6 7 9 4 3 7 7 8 9 5 1 9 5 4 8 6 2 5 7 5 3 4 3 5 7 5 1 5
6 2 1 8 3 4 6 1 6 1 9 5 2 1 3 2 8 2 2 8 8 1 4 1 3 7 6 7 1 5 4 5 6 3 4 1 4 2
5 6 9 9 1 5 3 2 1 3 1 6 2 8 3 9 6 4 7 1 1 9 9 5 7 5 6 5 6 7 8 5 1 2 1 8 8 7
1 7 1 7 4 9 8 7 3 8 7 1 9 8 6 1 7 6 9 8 6 1 1 1 9 1 2 7 1 3 1 8 3 5 7 9 3 1
6 5 7 4 1 8 8 3 9 2 6 2 1 5 2 2 6 3 1 6 7 1 5 6 5 4 6 8 4 8 6 5 7 4 1 9 2 4
9 6 9 9 1 1 6 1 9 3

baywatch5.txt

```
2 1 3 2 8 2 2 8 8 1 4 1 3 7 6 7 1 5 4 5 6 3 4 1 4 2 5 6 9 9 1 5 3 2 1 3 1 6
2 8 3 9 6 4 7 1 1 9 9 5 7 5 6 5 6 7 8 5 1 2 1 8 8 7 1 7 1 7 4 9 8 7 3 8 7 1
9 8 6 1 7 6 9 8 6 1 1 1 9 1 2 7 1 3 1 8 3 5 7 9 3 1 6 5 7 4 1 8 8 3 9 2 6 2
1 5 2 2 6 3 1 6 7 1 5 6 5 4 6 8 4 8 6 5 7 4 1 9 2 4 9 6 9 9 1 1 6 1 9 3 3 9
6 1 1 8 7 7 7 7 6 7 9 4 3 7 7 8 9 5 1 9 5 4 8 6 2 5 7 5 3 4 3 5 7 5 1 5 6 2
1 8 3 4 6 1 6 1 9 5
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? 7 ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? 7 ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? 2 ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? 9 1 ? ? ? ? ?
```

Dieses Beispiel besteht wieder aus 200 Landzungen. Es ist wie das Beispiel 4, aber mit weniger Fragezeichen. Es hat nur eine Lösung als Ergebnis mit der Verschiebung 150:

```
3 9 6 1 1 8 7 7 7 7 6 7 9 4 3 7 7 8 9 5 1 9 5 4 8 6 2 5 7 5 3 4 3 5 7 5 1 5
6 2 1 8 3 4 6 1 6 1 9 5 2 1 3 2 8 2 2 8 8 1 4 1 3 7 6 7 1 5 4 5 6 3 4 1 4 2
5 6 9 9 1 5 3 2 1 3 1 6 2 8 3 9 6 4 7 1 1 9 9 5 7 5 6 5 6 7 8 5 1 2 1 8 8 7
1 7 1 7 4 9 8 7 3 8 7 1 9 8 6 1 7 6 9 8 6 1 1 1 9 1 2 7 1 3 1 8 3 5 7 9 3 1
6 5 7 4 1 8 8 3 9 2 6 2 1 5 2 2 6 3 1 6 7 1 5 6 5 4 6 8 4 8 6 5 7 4 1 9 2 4
9 6 9 9 1 1 6 1 9 3
```

baywatch6.txt

```
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2
3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 1 1 2 1 2 3 4 1 2 3 4 5
6 7 8 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1
2 3 4 5 6 7 8 9 1 2 3 4 5
1 1 ? 1 ? ? ? 1 ? ? ? ? ? ? ? 1 ? ? ? ? ? ? ? ? ? ? ? ? ? ? 1 ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? 1 ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ? ? ? ? ?
```

Dieses Beispiel besteht aus 127 Fragezeichen. Es hat wieder nur eine Lösung als Ergebnis mit der Verschiebung 64:

```
1 1 2 1 2 3 4 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 1 2 3 4 5 6 7
8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 1 2 3 4 5 6 7 8 9 1 2 3 4
5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6
7 8 9 1 2 3 4 5 6 7 8 9 1
```

J2.3 Bewertungskriterien

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

- (1) [-1] **Dokumentation sehr unverständlich bzw. unvollständig**.

- (2) [-1] **Vorgegebenes Dateneingabeformat mangelhaft umgesetzt**:
  Das vorgegebene Eingabeformat sollte eingehalten und in eine geeignete interne Darstellung umgesetzt werden. Insbesondere sind lineare Datenstrukturen wie Listen oder Arrays geeignet.
- (3) [-1] **Lösungsverfahren fehlerhaft**:
  Das Lösungsverfahren, mit dem die Reihenfolge der Landzungen bestimmt wird, sollte korrekt sein. Es ist in Ordnung, wenn das Verfahren nicht nur eine Lösung, sondern mehrere Lösungen ermittelt. Läuft das Verfahren in eine Endlosschleife, wenn es keine Lösung (keine passende Landzungen-Liste) gibt, so führt dies jedoch zu keinem Punktabzug, da es in allen vorgegebenen Beispielen mindestens eine Lösung gibt.
- (4) [-1] **Strategie für Vergleich der Landzungen-Liste mangelhaft / fehlt**:
  Es wurde eine Strategie angewandt, mit welcher sich zuverlässig die Vergleiche der Landzungen-Listen in der passenden Reihenfolge durchführen lassen.
- (5) [-1] **Verfahren bzw. Implementierung unnötig aufwendig**:
  Nur besonders umständliche Verfahren bzw. Implementierungen führen zu Punktabzug. Die Laufzeiteffizienz wird nicht bewertet.
- (6) [-1] **Programmausgabe der Landzungen-Listen mangelhaft**:
  Die Landzungen-Listen müssen wie in der Aufgabenstellung angegeben vollständig in der richtigen Reihenfolge vom Programm ausgegeben werden, d.h. mit der nördlichen Landzunge beginnend und dem Uhrzeigersinn folgend. Es genügt jedoch, wenn nur mit einzelnen Zeichen/Zahlen/Ziffern anstelle von Wörtern für Landzungen im Programm gearbeitet wird. Es muss auch nichts in Wörter umgewandelt werden.
- (7) [-1] **Beispiele fehlerhaft bzw. zu wenige (mind. 3/6)**:
  Es wird die Dokumentation der Ergebnisse von mind. 3 der vorgegebenen 6 Beispiele erwartet. Für jedes Beispiel muss nur eine mögliche Landkarte als Ergebnis ausgegeben werden. Werden mehrere Ergebnisse (also Landkarten) für ein Beispiel ausgegeben, müssen alle korrekt sein.

Aufgabe 1: Superstar

1.1 Lösungsidee

Wenn wir die Anfrage „Folgt Mitglied x Mitglied y“ an das soziale Netzwerk stellen, können wir aus der Antwort eines der beiden Mitglieder als Superstar ausschließen: Falls die Antwort „Ja“ ist, kann x kein Superstar sein, weil x jemandem folgt. Falls die Antwort „Nein“ ist, kann y kein Superstar sein, weil nicht alle Mitglieder y folgen.

Dies können wir uns zunutze machen, um alle bis auf ein Mitglied als Superstar auszuschließen. Falls das Netzwerk n Mitglieder hat, brauchen wir dafür $(n-1)$-Fragen, die noch nicht ausgeschlossene Mitglieder beinhalten. So verwenden wir für die verbliebenen Kandidaten folgende Suchfunktion *Suche-Superstar-Kandidat*:

```
function Suche-Superstar-Kandidat
   Input: Mitglieder
  Output: Superstar-Kandidat
Kandidaten-Liste = liste(Mitglieder)
while |Kandidaten-Liste| > 1 do
        x = erstes Mitglied der Kandidaten-Liste
        y = zweites Mitglied der Kandidaten-Liste
        if(folgt(x,y)) then
                entferne x aus der Kandidaten-Liste
        else
                entferne y aus der Kandidaten-Liste
        end
end
return Mitglied, das bis zuletzt in der Kandidaten-Liste verblieb
```

Falls das Netzwerk einen Superstar hat, so ist er auf jeden Fall das von dieser Suchfunktion zurückgegebene Mitglied, weil ein Superstar niemals von dem in der Funktion angewandten Verfahren aus der Liste der Kandidaten entfernt werden kann. Da in jedem Schleifendurchlauf ein Kandidat entfernt wird, terminiert die Schleife.

Um sicherzustellen, dass wir tatsächlich einen Superstar gefunden haben, müssen wir noch die Bedingung überprüfen, ob jedes andere Mitglied dem Superstar-Kandidaten folgt und der Superstar-Kandidat keinem anderen Mitglied folgt. Dies können wir wie folgt in der Prüffunktion *Überprüfe-Superstar-Kandidaten* bestimmen:

```
function Überprüfe-Superstar-Kandidaten
   Input: Superstar-Kandidat, Mitglieder
  Output: Superstar-Kandidat oder "Kein Superstar gefunden"
foreach x in Mitglieder \ Superstar-Kandidat do
        if(folgt(Superstar-Kandidat, x) or not(folgt(x, Superstar-Kandidat))
            then
                return "Kein Superstar gefunden"
        end
end
return Superstar-Kandidat
```

Für diese Überprüfung des Superstar-Kandidaten benötigen wir $2(n-1)$ weitere Fragen. Insgesamt müssen also $3(n-1)$ Fragen an das Netzwerk gestellt werden.

1.2 Verringerung der Anfragenzahl

Beim Überprüfen wiederholen wir Fragen, für die wir bereits bei der Suche eine Antwort erhalten haben. Wenn wir uns diese Antworten merken, können wir Anfragen sparen.

Dies lässt sich sogar noch weiter ausnutzen, indem wir bei der Suche die Anfragen möglichst gleichmäßig auf die Kandidaten verteilen. So ist sichergestellt, dass wir möglichst viel über unseren Kandidaten erfahren. Hierfür ist eine kleine Anpassung unseres Suchverfahrens notwendig.

Wir entfernen in jedem Schritt statt einen Kandidaten die beiden aktuellen Kandidaten aus der Liste und fügen den einen der beiden, der weiter Superstar sein könnte, hinten wieder an die Liste an (diese Datenstruktur nennt man auch Warteschlange bzw. Queue nach dem FIFO-Prinzip). So stellen wir sicher, dass kein Mitglied, das kein Superstar ist, aber erst spät während der Suche ausgesondert wird, viele Anfragen über sich verbraucht:

```
function Verbesserte-Suche-Superstar-Kandidat
Input: Mitglieder
Output: Superstar-Kandidat
Kandidaten-Liste = liste(Mitglieder)
while |Kandidaten-Liste| > 1 do
        x = entferne erstes Mitglied von Kandidaten-Liste
        y = entferne erstes Mitglied von Kandidaten-Liste
        if(folgt(x,y)) then
                hänge y hinten an Kandidaten-Liste an
        else
                hänge x hinten an Kandidaten-Liste an
        end
end
return Mitglied, das bis zuletzt in der Kandidaten-Liste verblieb
```

1.3 Beispiele

Für die online angegebenen Beispiele könnte das eben verbesserte Verfahren wie folgt ablaufen:

superstar1.txt

```
[1] Beantwortete Anfrage: Selena folgt Justin
[2] Beantwortete Anfrage: Hailey folgt Justin
Überprüfe Justin...
[3] Beantwortete Anfrage: Justin folgt Selena nicht.
Bereits bekannt: Selena folgt Justin
[4] Beantwortete Anfrage: Justin folgt Hailey nicht.
Bereits bekannt: Hailey folgt Justin
Superstar gefunden: Justin
```

superstar2.txt

```
[1] Beantwortete Anfrage: Turing folgt Hoare
[2] Beantwortete Anfrage: Dijkstra folgt Knuth nicht.
[3] Beantwortete Anfrage: Codd folgt Hoare nicht.
[4] Beantwortete Anfrage: Dijkstra folgt Codd nicht.
Überprüfe Dijkstra...
[5] Beantwortete Anfrage: Dijkstra folgt Turing nicht.
[6] Beantwortete Anfrage: Turing folgt Dijkstra
[7] Beantwortete Anfrage: Dijkstra folgt Hoare nicht.
[8] Beantwortete Anfrage: Hoare folgt Dijkstra
Bereits bekannt: Dijkstra folgt Knuth nicht
[9] Beantwortete Anfrage: Knuth folgt Dijkstra
Bereits bekannt: Dijkstra folgt Codd nicht
[10] Beantwortete Anfrage: Codd folgt Dijkstra
Superstar gefunden: Dijkstra
```

superstar3.txt

```
[1] Beantwortete Anfrage: Edsger folgt Jitse
[2] Beantwortete Anfrage: Jorrit folgt Peter nicht.
[3] Beantwortete Anfrage: Pia folgt Rineke nicht.
[4] Beantwortete Anfrage: Rinus folgt Sjoukje nicht.
[5] Beantwortete Anfrage: Jitse folgt Jorrit nicht.
[6] Beantwortete Anfrage: Pia folgt Rinus nicht.
[7] Beantwortete Anfrage: Jitse folgt Pia nicht.
Überprüfe Jitse...
[8] Beantwortete Anfrage: Jitse folgt Edsger
Kein Superstar gefunden.
```

superstar4.txt

```
[1] Beantwortete Anfrage: Hanna folgt Melker nicht.
[2] Beantwortete Anfrage: Liv folgt Ellen nicht.
[3] Beantwortete Anfrage: Ali folgt Lova
[4] Beantwortete Anfrage: Vide folgt Freja nicht.
[5] Beantwortete Anfrage: Melvin folgt Loke nicht.
[6] Beantwortete Anfrage: Sigge folgt Milton
[7] Beantwortete Anfrage: Sofia folgt Arvid
...
[77] Beantwortete Anfrage: Noel folgt Emilia nicht.
[78] Beantwortete Anfrage: Charlie folgt Folke
[79] Beantwortete Anfrage: Noel folgt Folke
Überprüfe Folke...
...
...
...
Bereits bekannt: Charlie folgt Folke
```

```
[227] Beantwortete Anfrage: Folke folgt Penny nicht.
[228] Beantwortete Anfrage: Penny folgt Folke
[229] Beantwortete Anfrage: Folke folgt Rut nicht.
[230] Beantwortete Anfrage: Rut folgt Folke
Superstar gefunden: Folke
```

1.4 Bewertungskriterien

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

- (1) [-1] **Dokumentation sehr unverständlich bzw. unvollständig**.
- (2) [-1] **Vorgegebenes Dateneingabeformat mangelhaft umgesetzt**:
  Das vorgegebene Eingabeformat sollte eingehalten und in eine geeignete interne Darstellung umgesetzt werden. Insbesondere sind Listen dafür geeignet.
- (3) [-1] **Lösungsverfahren fehlerhaft**:
  Das Lösungsverfahren, mit dem der Superstar bestimmt wird, sollte korrekt sein. Insbesondere ist vom Verfahren sicher zu stellen, dass der letztlich gefundene Kandidat wirklich alle Superstar-Eigenschaften erfüllt. Die Ermittlung des Superstars bereits während des Einlesens der Daten ist keine Aufgabenlösung.
- (4) [-1] **Strategie zur Verringerung der Anfragenzahl mangelhaft / fehlt**:
  Es wurde eine Strategie angewandt, durch die die benötigte Zahl der Anfragen verringert wird. Doppelte Anfragen führen nicht zu Punktabzug, jedoch wird für naive Brute-Force-Strategien mit quadratischem Aufwand (Beziehungenabfragen von jedem zu jedem Kandidaten) ein Punkt abgehogen. Eine Aussage zur Abschätzung der maximal notwendigen Anfragenzahl wird nicht erwartet.
- (5) [-1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**:
  Nur besonders umständliche oder ineffiziente Verfahren bzw. Implementierungen führen zu Punktabzug.
- (6) [-1] **Programmausgabe der Superstar-Suche nicht nachvollziehbar**:
  Die Ausgabe der Superstar-Suche muss nachvollziehbar sein; eine reine Ja/Nein-Antwort ist zu wenig, aber es genügt, wenn die Anzahl der Abfragen und ggf. der gefundene Superstar ausgegeben werden. Am besten ist, wenn das Programm den gesamten Suchlauf ausgibt. Auch wenn es keinen Superstar gibt, so sollte das Programm dies entsprechend ausgeben.
- (7) [-1] **Beispiele fehlerhaft bzw. zu wenige (mind. 1/3 und 1/1)**:
  Es wird die Dokumentation der Ergebnisse von mind. 1 der 3 vorgegebenen Beispiele 1, 2 und 4 erwartet sowie das Ergebnis des Beispiels 3 (kein Superstar).

Aufgabe 2: Twist

2.1 Twisten

Lösung

Wir wollen Elvis helfen, seine Texte zu twisten. Dazu müssen wir uns überlegen, wie wir möglichst einfach die inneren Zeichen (also alle Zeichen außer dem Ersten und dem Letzten) eines Wortes mischen können. Dabei wollen wir idealerweise nicht nur irgendwelche Twists finden, sondern eine neue, zufällige Anordnung der inneren Zeichen in den getwisteten Wörtern mit gleicher Wahrscheinlichkeit erreichen.

Der erste Algorithmus, der einem einfallen mag, besteht daraus, alle zu mischenden Zeichen in eine Liste zu packen und dann solange zufällig Elemente aus dieser Liste zu entnehmen, bis die Liste leer ist. In Pseudocode könnte das ungefähr so aussehen:

```
1 mischen(wort w mit Länge n):
2    zeichen := {w[2], w[3], ..., w[n − 1]}
3    ergebnis := w[0]
4    solange zeichen nicht leer:
5       i := zufälliger Index in zeichen
6       ergebnis += zeichen[i]
7       entferne i in zeichen
8    ergebnis += w[n]
9    return Ergebnisse
```

Das hat allerdings zwei entscheidende Nachteile:

1. Man benötigt eine separate Liste für die zu mischenden Zeichen, was zusätzlichen Speicherplatz einnimmt.
2. Aufgrund der üblichen Implementationen einer Liste (Array-Liste und verkettete Liste) hat dieser Algorithmus eine quadratische Laufzeit.

Es gibt mit dem Fisher-Yates-Shuffle[1] jedoch einen Algorithmus, welcher beide Nachteile nicht hat. Er führt zufällige Vertauschungen aus, sodass die Eingabe am Ende gemischt ist[2]. In Pseudocode:

```
1 fisherYates(wort w mit Länge n):
2    für alle i von n − 1 bis 2:
3       j := Zufallszahl zwischen 2 und i
4       vertausche w[i] und w[j]
5    return w
```

Dieser Algorithmus gilt als Standardalgorithmus für das Mischen von Arrays und hat gegenüber unserem ersten Algorithmus wie bereits erwähnt zwei Vorteile:

1. Der Algorithmus arbeitet innerhalb der Liste („in-place“), benötigt also keine zusätzliche Liste.

[1] https://de.wikipedia.org/wiki/Zufällige_Permutation#Fisher-Yates-Verfahren

[2] Dabei sind alle Permutationen sogar gleich wahrscheinlich, was hier jedoch nicht erforderlich wäre

2. Der Algorithmus hat eine lineare Laufzeit, da nur $n-2$ Vertauschungen für jedes Wort der Länge n ausgeführt werden.

Zu beachten sind hierbei noch Randfälle:

- Wörter der Länge 1 oder 2 haben keine inneren Zeichen. Sie sollten deshalb durch das Twisten nicht verändert werden. Ebenso bleiben Wörter der Länge 3 mit nur einem inneren Zeichen unverändert. Wörter der Länge 4 haben zwei innere Zeichen, die entweder miteinander vertauschen werden können oder nicht. In all diesen Fällen ist das Enttwisten der Wörter kein Problem.
- Zahlen, Interpunktionszeichen, Leerzeichen und andere Zeichen müssen gesondert behandelt werden. So sollten z.B. Zahlen nicht getwistet werden und Interpunktionszeichen sollten nicht als Teil von Wörtern behandelt werden.

Um also einen Text zu twisten, werden die folgenden Schritte benötigt:

1. Die Eingabe wird eingelesen und durch einen geeigneten regulären Ausdruck in einzelne Wörter aufgespalten[3].
2. Wörter, die nur aus Buchstaben bestehen (also z.B. keine Jahreszahlen), werden mithilfe des Fisher-Yates-Verfahrens getwistet.
3. Die Wörter werden wieder aneinandergehängt und zusammen mit allen anderen Zeichen in der ursprünglichen Reihenfolge ausgegeben.

Beispiele

Für die online vorgegebenen vier Beispieltexte ergibt das Twisten folgende Ergebnisse:

Eingabe	Ausgabe
twist2.txt	Hat der atle Heneexstemir scih doch enaiml wgbeegeebn! Und nun seolln senie Gietesr auch ncah meienm Wielln leben. Seine Wrot und Wreke mrket ich und den Bucrah, und mit Gteäkrtsissee tu ich Wneudr auch.
twist3.txt	Ein Rarsuaetnt, wlhcees a la ctrae arebtiet, beitet sein Abgenot onhe eine vehorr fggteetelse Mfneeoülgerinhe an. Duardch haben die Gätse zwar mehr Slapeurim bei der Whal ihrer Seisepn, für das Rnatesuart etenhetsn jceodh zhtcäisulzer Awanufd, da weengir Pinahrigsshlcuenet vnarohden ist.
twist4.txt	Atsuuga Ada Byron King, Cnotuses of Lcelaove, war enie bhirtsice Agelide und Mktiiaehtmearn, die als die esrte Pomgriearremirn ürbapheut glit. Btreeis 100 Jarhe vor dem Afmuomken der etrsen Parishoerrgamprcemn enasrn sie enie Rehcen-Mnehicak, der egiine Kpnetoze mdoenerr Prmcmgsraaeeorirhpn vaeowhnrgm.
twist5.txt	Aicle fing an scih zu lwgenliaen; sie saß shocn lnage bei iehrr Scshteewr am Ufer und htate nhtics zu thun. Das Buch, das irhe Sehscwter las, gfieel ihr nihct; dnen es wrean wdeer Bidelr ncoh Gäpscerhe drain. „Und was nüzten Bceühr," dahtce Acile, „ohne Beidlr und Gcsähpree?" [...]

[3]Die meisten Sprachen bieten Möglichkeiten zum Umgang mit regulären Ausdrücken („RegEx"). Damit lassen sich Wortgrenzen ganz einfach mit dem regulären Ausdruck „\b" erkennen, für weitere Informationen siehe z. B. http://www.regexe.de/hilfe.jsp

2.2 Enttwisten

Lösung

Die zweite Aufgabe ist etwas komplizierter: Wir haben ein Wörterbuch zur Verfügung und sollen einen Text mit Hilfe dieses Wörterbuchs wieder enttwisten. Dabei stellt sich die Frage: Wie kann man ein getwistetes Wort möglichst einfach und effizient einem Eintrag im Wörterbuch zuordnen? Dazu müssen wir uns erst einmal klar machen, was ein getwistetes Wort mit dem Originalwort gemeinsam hat:

1. Der erste und der letzte Buchstabe im Wort sind identisch, da sie beim Twisten unverändert bleiben.
2. Beide Wörter haben die gleiche Länge und beinhalten dieselben inneren Zeichen, nur in einer anderen Reihenfolge.

So passt „Prarmgom“ mit dem Originalwort „Programm“ überein, denn beide beginnen mit „P“, enden mit „m“ und enthalten im Inneren die Zeichen „r“ (2 mal), „o“, „g“, „a“ und „m“. „Pmarmgor“ (endet auf „r“) oder „Prermgom“ (enthält ein „e“) würden demnach nicht zu „Programm“ passen.

Man könnte natürlich zu jedem eingegebenen Wort alle möglichen Permutationen der inneren Zeichen generieren und im Wörterbuch suchen; da das allerdings zu einer Laufzeit von $\mathcal{O}(n!)$ führt, ist dieser Ansatz gerade bei längeren Wörtern nicht praktikabel (schon das Wort „Informatiker“ hat über 3,6 Millionen mögliche Twists).

Um eine effizientere Lösung zu finden, machen wir uns folgende Dinge klar:

- Das Twisten ist symmetrisch. Beispiel: Da „Prarmgom“ ein Twist von „Programm“ ist, ist „Programm“ auch ein Twist von „Prarmgom“.
- Das Twisten ist transitiv, also wenn A ein Twist von B und B ein Twist von C ist, dann ist A auch ein Twist von C. Beispiel: Da „Prarmgom“ ein Twist von „Programm“ ist und „Programm“ ein Twist von „Pmargorm“ ist, ist „Prarmgom“ auch ein Twist von „Pmargorm“.

Diese beiden Eigenschaften können wir uns zu nutze machen, indem wir für jedes Wort eine Art „Basisform“ berechnen[4]. Am einfachsten lässt sich dies bewerkstelligen, wenn wir die inneren Zeichen einfach sortieren (diese Funktion nennen wir im folgenden h):

```
1 h(wort w):
2     inner := sort(w[2], w[3], ..., w[n − 1])
3     return (w[1] als Kleinbuchstabe) + inner + w[n]
```

Wir müssen uns außerdem darüber Gedanken machen, wie wir Groß- und Kleinschreibung behandeln. Ein am Anfang eines Satzes stehendes „Dsas“ entspricht sicherlich dem im Wörterbuch enthaltenen „dass“, obwohl sie nach unserer bisherigen Auffassung streng genommen keine Twists voneinander sind. Der einfachste Weg damit umzugehen, ist es, die Groß- und Kleinschreibung des ersten Zeichens bei der Berechnung der Basisform einfach zu ignorieren.

[4] Man kann hier auch von einer Hashfunktion sprechen, also einer Funktion, die eine große Menge an Eingaben auf eine kleine Menge an Ausgaben abbildet. Dabei bekommen „gleiche“ (also in unserem Fall ineinander twistbare) Wörter auch gleiche Hashwerte.

Insgesamt ist für unser Beispielwort „Programm“ dann $h(\text{"}Programm\text{"}) = \text{"}pagmorrm\text{"}$ und auch $h(\text{"}Prarmgom\text{"}) = \text{"}pagmorrm\text{"}$. Da die beiden Basisformen gleich sind, wissen wir dank Symmetrie und Transitivität auch, dass „Prarmgom“ ein Twist von „Programm“ ist.

Genau diesen „Umweg“ über das innen sortierte Wort können wir nun benutzen, um einen Index aus dem uns zur Verfügung gestellten Wörterbuch aufzubauen. Wir benötigen dafür eine Lookup-Datenstruktur (in den meisten Sprachen als Hashtabelle implementiert), in der wir jedes Wort w mit $h(w)$ als Schlüssel abspeichern. Das könnte dann folgendermaßen aussehen:

Schlüssel	Werte
...	...
pagmorrm	programm
aektur	akuter, akteur
hadelnte	handelte, haltende
haefr	hafer
...	...

Um nun ein Wort w zu enttwisten, müssen wir einfach nur seine Basisform $h(w)$ berechnen und im Index nachschauen, welche Wörter aus dem Wörterbuch dieselbe Basisform haben. So würden wir beim Wort „Prrmgoam“ die Basisform $h(\text{"}Prrmgoam\text{"}) = \text{"}pagmorrm\text{"}$ erhalten und über den Index dann das Wörterbuchwort „Programm“ finden. Dabei muss man auf die entsprechende Behandlung der Fälle achten, in denen der Schlüssel gar nicht oder mehrfach im Index vorkommt. Außerdem muss auch hierbei wieder auf Sonderzeichen geachtet werden.

Insgesamt müssen wir also beim Enttwisten, nachdem wir das Wörterbuch eingelesen und indiziert haben, wie beim Twisten die Eingabe in einzelne Wörter aufteilen, dann diese mit Hilfe ihrer Basisform enttwisten und die resultierenden Wörter in der ursprünglichen Reihenfolge ausgeben.

Laufzeitverhalten

Sei m die Länge des Wörterbuchs und n die Länge der Eingabe. Der erste Teil des Enttwisten-Algorithmus liest das Wörterbuch ein und indiziert es. Dafür werden die Basisformen berechnet (wofür wir sortieren müssen, was $\mathcal{O}(w \log w)$ Zeit benötigt) und diese in eine Lookup-Datenstruktur eingetragen. Mit der Annahme, dass diese Datenstruktur als Hashtabelle implementiert ist, ergibt sich dadurch eine Laufzeit von $\mathcal{O}(m \log m)$, da das Wörterbuch potentiell aus einem einzigen Wort der Länge m bestehen kann.

Für das Enttwisten müssen wir wieder für alle Wörter der Eingabe die Basisform berechnen und im Index danach suchen. Das ergibt auch hier eine Laufzeit von $\mathcal{O}(n \log n)$, womit das gesamte Verfahren eine Laufzeit von $\mathcal{O}(m \log m + n \log n)$ besitzt.

Unter der Annahme, dass die Wörter aus einer natürlichen Sprache stammen und damit in ihrer Länge konstant beschränkt sind, fallen die logarithmischen Faktoren weg; die Laufzeit ist in diesem Fall linear in m und n: $\mathcal{O}(m+n)$.

Beispiele

Für das auf dem Aufgabenblatt enthaltene Beispiel „twist1“ sowie drei weitere Beispiele ergibt das Enttwisten folgende Ergebnisse:

Eingabe	Ausgabe
twist1	der Twsit? (englisch tiwst? = drehung, verdrehung) war ein Mdaotenz? im 4/4-takt, der in den [frühen,führen] 1960er jahren populär wurde und zu rock'n'Roll?, Ryhthm? and blues oder spezieller Twsit?-musik getanzt wird.
beispiel2	es existiert ein verfahren, womit man einen text maschinell unlesbar machen kann, sodass ihn ein mensch trotzdem noch lesen kann. dabei werden die zeichen, bis auf das erste und das letzte, jedes einzelnen wortes in eine beliebige reihenfolge gebracht. dadurch wird es einem computer erschwert, den text mithilfe von wortlisten zu analysieren, gleichzeitig kann ein mensch den text beim lesen trotzdem noch intuitiv verstehen.
beispiel3	das Z ist der letzte buchstabe des lateinischen alphabets. er kommt in deutschen texten relativ selten vor, findet aber in [anreden,anderen] sprachen, wie zum beispiel englisch oder polnisch, größere verwendung. er ist verwandt mit dem griechischen zeta und hat sich ursprünglich aus dem symbol einer stichwaffe entwickelt.
beispiel4	beim maschinellen lernen wird in der regel versucht, automatisiert aus einer menge an beispielen einen aneleemgin? zusammenhang herauszufinden. die theoretische grundlage dieser verfahren bildet das berechnen von empirischen wahrscheinlichkeitsverteilungen, aus denen sich dann die gewünschten informationen [schleißen,schließen] lassen.

2.3 Bewertungskriterien

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

- (1) [-1] **Dokumentation sehr unverständlich bzw. unvollständig**.
- (2) [-1] **Vorgegebenes Dateneingabeformat mangelhaft umgesetzt**:
  Das vorgegebene Eingabeformat sollte eingehalten und in eine geeignete interne Darstellung umgesetzt werden. Insbesondere sind lineare Datenstrukturen für die Darstellung der Texte als Folgen von Wörtern geeignet. Auch sollten die Groß- und Kleinschreibung von Wörtern berücksichtigt werden sowie die Interpunktionszeichen und andere (Sonder-)Zeichen im Text korrekt behandelt werden.
- (3) [-1] **Lösungsverfahren fehlerhaft**:
  Die Verfahren zum Twisten und Enttwisten sollten beide korrekt sein. Die Anfangs- und Endbuchstaben der Wörter sollten an ihrer ursprünglichen Position bleiben, alle anderen Buchstaben im Inneren der Wörter müssen zufällig gemischt werden. Sollte statt einer zufälligen Umordnung (wie gemäß der Aufgabenstellung) die inneren Buchstaben nur umsortiert werden, so gibt dies ohne ausreichende Begründung (z. B. hinsichtlich des ähnlichen Informationsverlust im Wort) einen Punkt Abzug. Es ist grundsätzlich in Ordnung, wenn z.B. „Dass“ und „dass“ wegen Groß-/Kleinschreibung beim Enttwisten als verschieden aufgefasst werden. Eine Aussage über die Qualität des Verfahrens zum Enttwisten wird nicht erwartet.
- (4) [-1] **Strategie für Ähnlichkeitsvergleich mangelhaft / fehlt**:
  Es wurde eine Strategie für das Enttwisten angewandt, die den Ähnlichkeitsvergleich zweier Wörter sinnvoll bewerkstelligt.

- (5) [-1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**:
  Nur besonders umständliche oder ineffiziente Verfahren bzw. Implementierungen führen zu Punktabzug. Da der sehr einfache Algorithmus zum Twisten mittels der Datenstruktur einer Liste quadratische Laufzeit besitzt, sollte das Verfahren fürs Twisten auf keinen Fall langsamer sein. Fürs Enttwisten sollte das Generieren und Überprüfen aller Permutationen vermieden werden. Das Durchsuchen eines Wörterbuchs in linearer Laufzeit ist jedoch in Ordnung.
- (6) [-1] **Programmausgabe ungeeignet**:
  Eine geeignete Ausgabe der Ergebnistexte des Twisten bzw. Enttwisten sollte vorhanden sein. Wenn ein Wort nicht enttwistet werden konnte, muss dies in der enttwisteten Lösung nicht kenntlich gemacht sein.
- (7) [-1] **Beispiele fehlerhaft bzw. zu wenige (mind. 2/4 und 1)**:
  Es wird die Dokumentation der Ergebnisse von mind. 2 der 4 vorgegebenen Beispiele erwartet. Außerdem muss mind. 1 Beispiel für das Entwisten aussagekräftig genug sein, um einen Punktabzug zu vermeiden. Das 5. Beispiel darf aufgrund seiner Textlänge stark verkürzt in der Dokumentation enthalten sein.

Aufgabe 3: Voll Daneben

3.1 Problemstellung

Al Capone Junior bekommt von den Teilnehmern eine Liste mit n Zahlen. Wir ordnen diese Liste der Größe nach und nennen ihre Einträge $t_1, \ldots, t_n$. Das Ziel von Al ist es, m Zahlen $d_1, \ldots, d_m$ so zu wählen, dass die Summe der Abstände von t_i zu den jeweils nächsten d_j möglichst gering wird. In der Aufgabe war zwar nur $m = 10$ verlangt, aber wir lösen dies allgemein.

Eine Lösung von Al heißt *optimal*, wenn er seine Auszahlung nicht mehr weiter verringern kann, egal wie er seine Zahlen wählt. Der Algorithmus in dieser Musterlösung berechnet immer eine optimale Lösung für Al.

Beachte, dass es mehrere optimale Lösungen geben kann. Angenommen Al darf nur eine Zahl bestimmen. Haben wir zwei Teilnehmer mit den Zahlen 10 und 20, so kann Al jede beliebige Zahl zwischen 10 und 20 wählen und die Ausschüttung beträgt in diesen Fällen genau 10 Taler, was auch optimal ist. Wählt Al dagegen eine Zahl unter 10 oder über 20, so liegt seine Auszahlung über 10 Taler.

3.2 Erste Ideen

Ein Problem ist, dass Al sehr viele Wahlmöglichkeiten für jede einzelne seiner Zahlen hat. Um diese Möglichkeiten etwas einzuschränken, benutzen wir folgende Information:

Für Al gibt es immer eine optimale Lösung, die ausschließlich Zahlen der Teilnehmer verwendet.

Wir wollen uns überlegen, wieso dies der Fall ist. Nehmen wir an, Als Lösung enthält eine Zahl d, die von keinem Teilnehmer gewählt wurde. Ist diese Zahl d kleiner als alle Teilnehmerwerte, so kann keine optimale Lösung vorliegen. Wir könnten dann einfach d in Richtung der Teilnehmerwerte verschieben und erhalten eventuell eine bessere Lösung (falls d die Auszahlung tatsächlich beeinflusst), jedenfalls keine schlechtere. Das Gleiche passiert natürlich auch, wenn d größer als alle Teilnehmerwerte ist. Der interessante Fall ist somit, wenn Als Zahl d zwischen zwei Teilnehmerwerten t_i und t_{i+1} liegt.

Liegt eine Teilnehmerzahl t_j näher an d als alle anderen Zahlen von Al, so sagen wir, dass t_j von d *getroffen* wird. Die Liste der von d getroffenen Teilnehmerwerte können wir in zwei Gruppen aufteilen, nämlich diejenigen Teilnehmerwerte, die kleiner als d sind und diejenigen, die größer als d sind. Zur Abkürzung schreiben wir:

$$L(d) = \text{Liste der von } d \text{ getroffenen Teilnehmerwerte, die kleiner als } d \text{ sind.}$$
$$R(d) = \text{Liste der von } d \text{ getroffenen Teilnehmerwerte, die größer als } d \text{ sind.}$$

Je nachdem, ob die Anzahl $|L(d)|$ oder $|R(d)|$ größer ist, verschieben wir Als Zahl d entweder auf den Wert t_i oder t_{i+1}. Alle anderen seiner Zahlen lassen wir dabei gleich. Durch die Verschiebung von d verändern sich auch die Listen $L(d)$ und $R(d)$.

Im Fall $|L(d)| \geq |R(d)|$ verschieben wir d auf t_i. Diese Verschiebung hat folgende Auswirkungen:

a) Für alle Teilnehmer aus $L(d)$ verringert sich die Auszahlung um $d - t_i$.

b) Für alle Teilnehmer aus $R(t_i)$ erhöht sich die Auszahlung um $d - t_i$.

c) Beim Wechsel von $L(d)$ zu $L(t_i)$ kommen eventuell neue Teilnehmer dazu. Für diese verringert sich die Auszahlung um einen Betrag zwischen und 0 und $d - t_i$.

d) Beim Wechsel von $R(d)$ zu $R(t_i)$ fallen eventuell alte Teilnehmer weg. Für diese erhöht sich die Auszahlung um einen Betrag zwischen und 0 und $d - t_i$.

e) Für alle restlichen Teilnehmer bleibt die Auszahlung konstant.

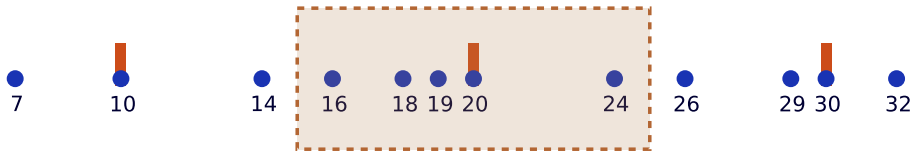
Wegen der Bedingung $|L(d)| \geq |R(d)|$ sind die zusätzlichen Auszahlungen bei b) und d) auf keinen Fall größer als die Einsparungen in a) und c). Durch die Verschiebung verschlechtert sich also Als Lösung nicht. Es kann zwar passieren, dass Al lediglich eine andere gleich gute Anordnung findet, aber in der Regel wird die neue Anordnung sogar günstiger sein.

Im Fall $|L(d)| < |R(d)|$ verschieben wir d auf t_{i+1}. Hier treten die gleichen Auswirkungen wie vorhin auf, außer dass jetzt alles spiegelverkehrt ist.

Beispiel:

Die Teilnehmer wählen die Zahlen 7, 10, 14, 16, 18, 19, 20, 24, 26, 29, 30 und 32. Al soll drei Zahlen wählen. Diese wählt er als 10, 23 und 30.

Die blauen Punkte sind die Zahlen der Teilnehmer, die roten Striche sind Als Zahlen. Der eingefärbte Bereich wird von der Zahl 23 getroffen. Wir sehen, dass auf der linken Seite mehr Zahlen von 23 getroffen werden als auf der rechten Seite. Folglich verschieben wir die Zahl 23 auf den linken Nachbar 20.



Durch die Verschiebung $23 \to 20$ wird jetzt 16 getroffen und 26 nicht mehr. Die Auszahlungen ändern sich wie folgt:

Teilnehmer	7	10	14	16	18	19	20	24	26	29	30	32
Ausz. vorher	3	0	4	6	5	4	3	1	3	1	0	2
Ausz. nachher	3	0	4	4	2	1	0	4	4	1	0	2
Änderung	0	0	0	-2	-3	-3	-3	+3	+1	0	0	0

Insgesamt zahlt Al so 7 Taler weniger.

3.3 Zerlegung in Teilprobleme

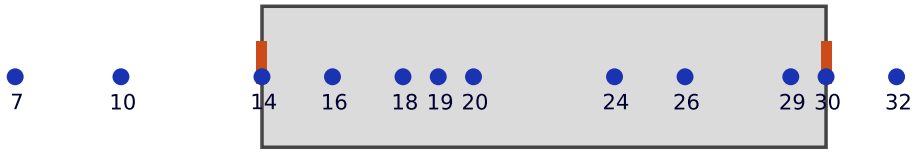
Dank der vorherigen Überlegung können wir davon ausgehen, dass Al alle seine Zahlen aus der Liste der Teilnehmer auswählt. Dies reduziert schon mal die Anzahl der Möglichkeiten, aber es sind immer noch zu viele. Deswegen brauchen wir weitere Ideen.

Oft lassen sich Probleme in der Informatik dadurch lösen, indem man sie in kleinere *Teilprobleme* zerlegt, diese jeweils für sich löst und anschließend die Teilergebnisse wieder zusammensetzt. Dieser Ansatz funktioniert auch hier, wenn wir uns klarmachen, was unsere Teilprobleme sind.

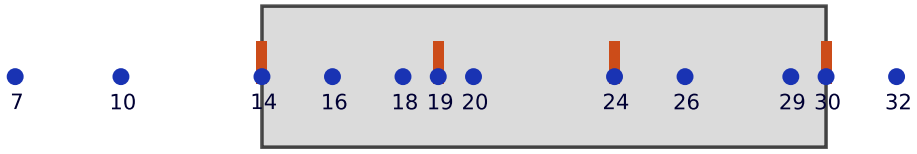
Nehmen wir an, Al hat sich schon zwei Teilnehmerwerte t_i und t_j auf irgendeine Art und Weise ausgesucht. Zwischen t_i und t_j sollen sich bislang noch keine weiteren Zahlen von Al befinden. Nun möchte sich Al l weitere Zahlen in diesem Intervall $[t_i, t_j]$ aussuchen. Unser Teilproblem besteht nun darin, diese l Zahlen so zu berechnen, dass die Summe der Auszahlungen für alle Teilnehmer innerhalb dieses Intervalls $[t_i, t_j]$ minimal wird (die anderen Teilnehmer interessieren uns erstmal nicht). Haben wir l solcher passenden Zahlen gefunden, so sprechen wir von einer *optimalen Teillösung* für $[t_i, t_j]$ mit l Zahlen.

Beispiel:

Zwischen 14 und 30 sollen zwei weitere Zahlen bestimmt werden:



Eine optimale Teillösung schaut so aus:



Anfangsbelegung:

Bei den Teillösungen gehen wir davon aus, dass Al bereits zwei Zahlen gewählt hat. Da wir am Anfang noch keine zwei Zahlen zur Verfügung haben, behelfen wir uns eines kleinen Tricks: Wir fügen zu der Zahlenliste der Teilnehmer noch zwei Dummyzahlen $-\infty$ und ∞ hinzu (in der Praxis gingen auch große Zahlen wie -1000 und 2000). Al bekommt ebenfalls zwei weitere Zahlen, die er allerdings auf diese Dummyzahlen festlegen muss. Diese Operation hat keinen Einfluss auf die Auszahlung, allerdings stehen uns jetzt zwei Ausgangszahlen zur Verfügung.

3.4 Optimalitätsprinzip

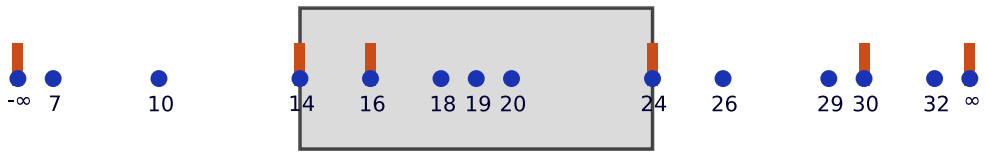
Ein *Optimalitätsprinzip* liegt vor, wenn sich eine optimale Gesamtlösung aus mehreren optimalen Teillösungen zusammensetzen lässt. In solchen Fällen findet man oft effiziente Algorithmen, wie auch hier:

Wählen wir aus einer optimalen Lösung Als $d_1, \ldots, d_m$ zwei Zahlen d_i und d_j aus, so minimiert die Verteilung seiner l Zahlen im Intervall $[d_i, d_j]$ die Summe der Auszahlungen an die Teilnehmer in diesem Intervall $[d_i, d_j]$.

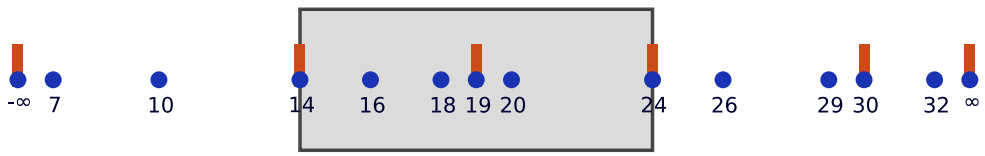
Aber wieso können wir das Optimalitätsprinzip hier überhaupt anwenden? Finden wir innerhalb des Intervalls $[d_i, d_j]$ eine bessere Verteilung mit l Zahlen (wobei d_i und d_j weiterhin gesetzt bleiben), so können wir in der Gesamtlösung ebenfalls diesen Bereich durch die bessere Teillösung austauschen. Die Teilnehmer im Intervall $[d_i, d_j]$ bekommen durch den Austausch insgesamt eine geringere Auszahlung. Für die Teilnehmer außerhalb des Intervalls $[d_i, d_j]$ bleibt die Auszahlung durch den Austausch unbeeinflusst. Für solche Teilnehmer ist Als nächste Zahl entweder höchstens d_i oder mindestens d_j, jedoch nie eine Zahl zwischen d_i und d_j.

Beispiel:

Folgende Teillösung im Bereich $[14, 24]$ ist nicht optimal, statt der Zahl 16 sollte Al lieber die Zahl 19 wählen:



Im Bereich $[14, 24]$ wurde die suboptimale Teillösung durch eine verbesserte Variante ersetzt:



Durch diese Ersetzung sind die Gesamtkosten um 5 Taler gesunken.

3.5 Berechnung einer optimalen Lösung

Wir lesen die Zahlen der Teilnehmer ein und sortieren sie der Größe nach. Dabei fügen wir auch die beiden Dummyzahlen hinzu. Wir erhalten so eine Liste $t_1, \ldots, t_n$, wobei die Einträge t_1 und t_n unsere beiden Dummyzahlen sind.

Wenn sich Al mindestens so viele Zahlen aussuchen darf, als es Werte von Teilnehmern gibt, können wir Al einfach diese Werte auswählen lassen. Alle Teilnehmer gehen dann leer aus.

Ansonsten berechnen wir die einzelnen Teillösungen. Hierbei müssen wir nicht alle Intervalle $[t_i, t_j]$ behandeln. Stattdessen genügt es, sich auf diejenigen Intervalle zu beschränken, die von einem t_i bis ganz zum Ende t_n reichen. Wir brauchen folgende Variablen:

LÖSUNG(i,l) speichert die optimale Teillösung für die Teilnehmer t_i bis t_n mit l Zahlen.

KOSTEN(i,l) gibt die Auszahlung von LÖSUNG(i,l) an die Teilnehmer t_i bis t_n an.

KOSTENABSCHNITT(i,j) gibt die Summe der Auszahlungen an, die an Teilnehmer t_i bis t_j ausgezahlt werden müssen, sofern Al die Zahlen t_i und t_j festgelegt hat und es dazwischen keine weiteren Zahlen von ihm mehr gibt.

Zunächst berechnen wir die Tabelle KOSTENABSCHNITT und legen die Anfangsbedingungen

$$\text{LÖSUNG}(i,0) = \textit{leer} \text{ und } \text{KOSTEN}(i,0) = \text{KOSTENABSCHNITT}(i,n)$$

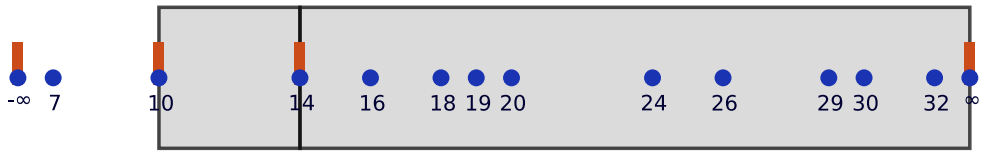
fest. Um LÖSUNG(i,l) für $l > 0$ zu berechnen, führen wir folgende Schritte aus:

1. Wir lassen j alle Zahlen von $i+1$ bis $n-1$ durchlaufen.
2. Für jedes j teilen wir das Intervall $[t_i, t_n]$ in die Teilintervalle $[t_i, t_j]$ und $[t_j, t_n]$ auf.
3. Die l Zahlen im Intervall $[t_i, t_n]$ werden wie folgt auf die Teilintervalle verteilt:
   a) Die Zahl t_j wird für die Teillösung gewählt.
   b) Im Innern von $[t_i, t_j]$ wird keine weitere Zahl gewählt. Die Kosten dieses Abschnittes sind damit KOSTENABSCHNITT(i,j).
   c) Im Innern von $[t_j, t_n]$ werden $l-1$ weitere Zahlen gewählt. Die Wahl dieser Zahlen wird rekursiv durch LÖSUNG$(j, l-1)$ bestimmt. Die Kosten dieses Abschnitts sind KOSTEN$(j, l-1)$.
4. Auf diese Weise erhalten wir einen möglichen Kandidaten für LÖSUNG(i,l) mit Kosten KOSTENABSCHNITT(i,j) + KOSTEN$(j, l-1)$.
5. Unter allen diesen Kandidaten suchen wir uns den günstigsten heraus. Diesen speichern wir in LÖSUNG(i,l) und dessen Kosten in KOSTEN(i,l).

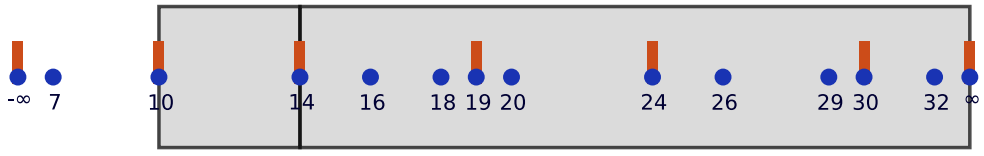
Die Gesamtlösung für $[t_1, t_n]$ mit m Zahlen erhalten wir dann durch Auslesen von LÖSUNG$(1,m)$ und deren Kosten aus KOSTEN$(1,m)$.

Beispiel:

Wir wollen LÖSUNG$(3,4)$ bestimmen, also eine Teillösung von $t_3 = 10$ bis $t_n = \infty$ mit vier weiteren Zahlen zwischen 10 und ∞. Dazu können wir das Intervall bei $j = 4$ aufteilen

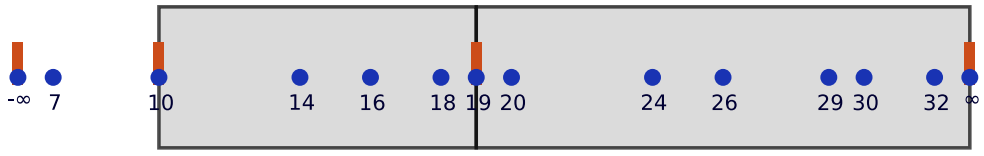


und anschließend LÖSUNG$(4,3)$ berechnen:

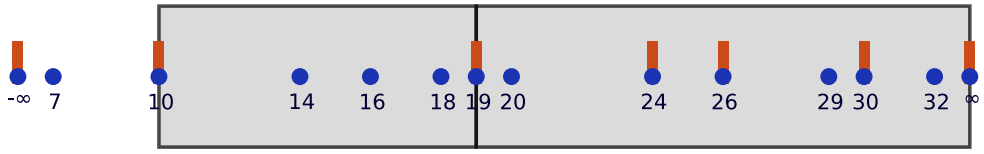


Es ist KOSTENABSCHNITT$(3,4) = 0$ und KOSTEN$(4,3) = 9$. Die Kosten im eingefärbten Bereich belaufen sich also auf 9 Taler.

Alternativ können wir das Intervall beispielsweise bei $j = 7$ aufteilen

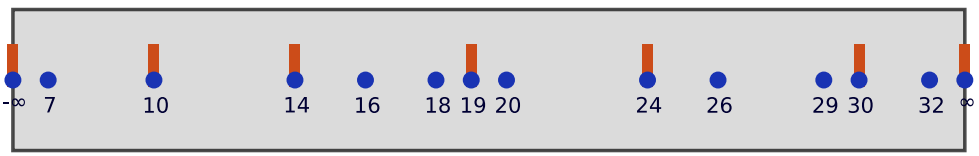


und anschließend LÖSUNG$(7,3)$ berechnen:



Es ist KOSTENABSCHNITT$(3,7) = 8$ und KOSTEN$(7,3) = 4$. Die Kosten im eingefärbten Bereich belaufen sich also auf 12 Taler.

Wir würden also die Aufteilung bei $j = 4$ wegen der geringeren Kosten bevorzugen. Natürlich müssen wir jetzt noch alle möglichen Aufteilungen überprüfen, aber es wird sich zeigen, dass die Aufteilung bei $j = 4$ tatsächlich optimal ist. Eine optimale Gesamtlösung für fünf Zahlen schaut so aus:



3.6 Kosten der einzelnen Abschnitte

Für unseren Algorithmus benötigen wir noch eine effiziente Methode, um alle Werte in der Tabelle KOSTENABSCHNITT zu bestimmen. Ein naiver Ansatz wäre es, für jedes Paar (i, j) die Summe

$$\text{KOSTENABSCHNITT}(i,j) = \sum_{k=i}^{j} \min\{t_k - t_i, t_j - t_k\}$$

zu berechnen. Dieser Ansatz führt zu einer Laufzeit von $\Theta(n^3)$ in diesem Zwischenschritt. Eine bessere Laufzeit erhalten wir, wenn wir zuerst Hilfssummen

$$\text{SUMME}(i,j) = \sum_{k \in \{i,\ldots,j\}} |t_k - t_i|$$

bestimmen. Summe(i, j) ist der Auszahlungsbetrag für t_i bis t_j, der entsteht, wenn die jeweiligen Teilnehmerzahlen von t_i getroffen werden. Diese Hilfssummen können wir iterativ berechnen:

$$\text{SUMME}(i,i) = 0$$

$$\text{SUMME}(i,j) = \begin{cases} \text{SUMME}(i,j-1) + t_j - t_i & \text{für } j > i \\ \text{SUMME}(i,j+1) + t_i - t_j & \text{für } j < i \end{cases}$$

Die Kosten für einen Abschnitt erhalten wir dann mit Hilfe der Formel

$$\text{KOSTENABSCHNITT}(i,j) = \text{SUMME}(i,h-1) + \text{SUMME}(j,h).$$

Der Wert h ist so bestimmt, dass t_{h-1} näher an t_i liegt sowie t_h näher an t_j.

Benutzt man für jedes Paar (i, j) eine Binärsuche, um dieses h zu finden, so erhält man eine Laufzeit von $\Theta(n^2 \cdot \log(n))$. Mit einem Trick geht es aber noch schneller: Kennen wir nämlich h für KOSTENABSCHNITT(i, j), so müssen wir das h für KOSTENABSCHNITT$(i, j+1)$ nur um einen nicht allzu großen Wert erhöhen. Berechnen wir für jedes i die KOSTEN(i, j) nach aufsteigenden j, so erzielen wir eine Laufzeit von $\Theta(n^2)$.

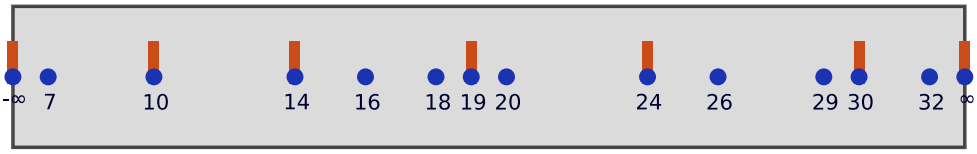
Algorithmus 1 Diese Funktion berechnet die Tabelle KOSTENABSCHNITT

```
function BERECHNEKOSTENFUERABSCHNITTE(t_1, ..., t_n)
    SUMME ← BERECHNESUMMEN(t_1, ..., t_n)
    for i von 1 bis n do
        h ← i
        for j von i + 1 bis n do
            while (t_h − t_i) < (t_j − t_h) do
                h ← h + 1
            end while
            KOSTENABSCHNITT(i, j) ← SUMME(i, h − 1) + SUMME(j, h)
        end for
    end for
    return KOSTENABSCHNITT
end function
```

3.7 Tricks zur Effizienzsteigerung

Speichern der Teillösungen: Prinzipiell ist es möglich, in LÖSUNG(i,l) eine komplette Zahlenliste der Teillösung zu speichern. Dies ist aber keine effiziente Vorgehensweise. Stattdessen speichern wir in LÖSUNG(i,l) lediglich einen Index j. Dieser Index j gibt die kleinste verwendete Zahl t_j der Teillösung in $[t_i,t_n]$ an, die nicht t_i ist. Diese Information reicht bereits aus, um daraus die komplette Gesamtlösung zu rekonstruieren. Hat Al für seine Lösung die i-te Teilnehmerzahl t_i bereits ausgewählt, so lautet seine nächste Zahl t_j mit $j =$ LÖSUNG(i,l), wobei l die Anzahl der noch unbestimmten Zahlen Als ist.

Schauen wir nochmal unsere optimale Lösung an:



Wir würden LÖSUNG$(1,5) = 3$ speichern, denn $t_3 = 10$ ist die erste Zahl. Die nächste Zahl wird durch LÖSUNG$(3,4) = 4$ mit $t_4 = 14$ angegeben, die übernächste durch LÖSUNG$(4,3) = 7$ mit $t_7 = 19$ und so weiter.

Dynamische Programmierung: In der Rekursion müssen wir LÖSUNG(i,l) mehrfach bestimmen. Es wäre nicht gerade schlau, diese Berechnung jedes Mal komplett neu auszuführen. Stattdessen berechnen wir die optimalen Teillösungen LÖSUNG(i,l) nur einmal und speichern dieses Ergebnis dann. Diese Technik ist auch unter dem Begriff der dynamischen Programmierung[5] bekannt. Dabei ist es wichtig, dass wir die Teillösungen LÖSUNG(i,l) in aufsteigenden Reihenfolge der Längen l berechnen, denn nur so können wir auf die bereits berechneten kürzeren Teillösungen LÖSUNG$(j,l-1)$ zurückgreifen.

3.8 Laufzeitüberlegungen

Der Algorithmus zur Bestimmung aller Teillösungen LÖSUNG(i,l) mittels dynamischer Programmierung hat eine Laufzeit von $\Theta(n^2 \cdot m)$. Es gibt nämlich insgesamt $n \cdot m$ Teillösungen, und zur Berechnung einer einzelnen Teillösung müssen wir im Schnitt etwa $\frac{n}{2}$ Kandidaten berücksichtigen. Das Sortieren der Teilnehmerwerte hat eine Laufzeit von $\Theta(n \cdot \log(n))$. Wie wir gerade gesehen haben, liegt die Berechnung aller KOSTENABSCHNITT(i,j) in $\Theta(n^2)$. Die beiden letzten Operationen sind also für große Eingaben laufzeittechnisch vernachlässigbar und die Gesamtlaufzeit liegt bei $\Theta(n^2 \cdot m)$.

3.9 Beispiele

Im Folgenden sind die Ergebnisse des Programms für obiges Beispiel sowie für die drei vorgegebenen Beispiele angegeben. Außerdem ist extra ein Beispiel angefügt, wie Al Verlust machen kann.

[5] https://de.wikipedia.org/wiki/Dynamische_Programmierung

Algorithmus 2 Berechnung der optimalen Lösung mittels dynamischer Programmierung

```
function BERECHNEOPTIMALELOESUNG(t_1, ..., t_n, m)
    Eingabe: Sortierte Teilnehmerwerte t_1, ..., t_n inkl. Dummies, Anzahl m für Al
    Ausgabe: Als Zahlen d_1, ..., d_m sowie deren Kosten

    ▷ Lege die Anfangsbedingungen fest.
    KOSTENABSCHNITT ← BERECHNEKOSTENFUERABSCHNITTE(t_1, ..., t_n)
    for i von 1 bis n do
        KOSTEN(i, 0) = KOSTENABSCHNITT(i, n)
    end for

    ▷ Berechne alle optimalen Teillösungen.
    for l von 1 bis m do
        for i von 1 bis n do
            KOSTEN(i, l) ← ∞
            for j von i + 1 bis n − 1 do
                if KOSTENABSCHNITT(i, j) + KOSTEN(j, l − 1) < KOSTEN(i, l) then
                    LÖSUNG(i, l) ← j
                    KOSTEN(i, l) ← KOSTENABSCHNITT(i, j) + KOSTEN(j, l − 1)
                end if
            end for
        end for
    end for

    ▷ Baue die Zahlenfolge für Al zusammen.
    i ← 1, AUSGABE ← leer
    for l von m bis 1 abwärts do
        i ← LÖSUNG(i, l)
        Füge t_i zu AUSGABE hinzu.
    end for
    return AUSGABE, KOSTEN(1, m)
end function
```

Obiges Beispiel

```
> python volldaneben.py beispiel-skizzen.txt 5
Zahlen der Teilnehmer:
 7 10 14 16 18 19 20 24 26 29 30 32
Al wählt die Zahlen:
 7 14 19 24 30
Einnahmen durch die Einsätze: 300 Taler
Auszahlungen an die Teilnehmer: 12 Taler
Gewinn für Al: 288 Taler
```

Beispiel 1

```
> python volldaneben.py beispiel1.txt
Zahlen der Teilnehmer:
 5 10 15 20 25 30 ... 990 995 1000
Al wählt die Zahlen:
 50 145 240 335 430 525 630 735 840 945
Einnahmen durch die Einsätze: 4975 Taler
Auszahlungen an die Teilnehmer: 4950 Taler
Gewinn für Al: 25 Taler
```

Beispiel 2

```
> python volldaneben.py beispiel2.txt
Zahlen der Teilnehmer:
 11 12 22 27 42 43 58 59 60 67 69 84 87 91 94 123 124 135 167 170 172
 178 198 211 226 229 276 281 305 313 315 324 327 335 336 362 364 367 368
 368 370 373 383 386 393 399 403 413 421 421 426 429 434 456 492 505 526
 530 537 539 540 545 567 582 584 586 649 651 676 690 729 736 739 750 754
 763 777 782 784 788 793 802 808 814 846 857 862 862 873 886 895 915 919
 925 926 929 932 956 980 996
Al wählt die Zahlen:
 59 172 315 368 421 539 651 777 862 929
Einnahmen durch die Einsätze: 2500 Taler
Auszahlungen an die Teilnehmer: 1924 Taler
Gewinn für Al: 576 Taler
```

Beispiel 3

```
> python volldaneben.py beispiel3.txt
Zahlen der Teilnehmer:
 20 40 60 80 100 100 120 120 140 160 160 180 200 220 220 240 240 240 240 260 260
 260 280 280 300 300 340 340 340 340 360 360 380 380 400 400 420 420 440 440 460
 460 460 480 480 500 520 520 520 520 520 520 540 540 540 560 580 580 580 580 600
 600 620 640 640 640 660 680 680 680 680 700 700 720 720 720 720 720 740 740 760
 780 780 800 800 820 840 840 860 860 860 880 900 900 920 920 960 980 980 1000
Al wählt die Zahlen:
 100 240 340 440 520 580 660 720 860 960
```

```
Einnahmen durch die Einsätze: 2500 Taler
Auszahlungen an die Teilnehmer: 2160 Taler
Gewinn für Al: 340 Taler
```

Extra-Beispiel für Als Verlust

Wir wählen uns 21 Zahlen zwischen 1 und 1000, so dass aufeinanderfolgende Zahlen genau Abstand 49 haben.

```
> python volldaneben.py beispiel-verlust.txt
Zahlen der Teilnehmer:
 1 50 99 148 197 246 295 344 393 442 491 540 589 638 687 736 785 834 883 932 981
Al wählt die Zahlen:
 1 50 99 148 197 344 491 638 785 932
Einnahmen durch die Einsätze: 525 Taler
Auszahlungen an die Teilnehmer: 539 Taler
Gewinn für Al: -14 Taler
```

Wie wir sehen, macht Al in diesem Fall Verlust. Auch ohne das Programm auszuführen kann man leicht den Grund sehen. Von den 21 Zahlen der Teilnehmer können nämlich nur 10 Zahlen von Al gewählt werden. Die restlichen 11 Zahlen haben mindestens einen Abstand von 49 zu den nächsten Zahlen Als. Damit muss Al mindestens $11 \cdot 49 = 539$ Taler auszahlen, aber seine Einnahmen betragen nur $21 \cdot 25 = 525$.

3.10 Alternative Lösungsansätze

Einleitung

Diese Aufgabe ermöglicht eine optimale Lösung in polynomieller Zeit, wie sie oben beschrieben wird. Diese ist allerdings nicht leicht zu finden. Im Folgenden werden einige Ansätze erläutert, welche mit heuristischen Vorgehensweisen zu guten oder schlechten Ergebnissen für die vorgegebenen Beispiele führen.

Wir erhalten als Eingabe eine variable Anzahl an Zahlen und benötigen zehn Zahlen für Al (Al-Zahlen), die Als Kosten minimieren. Für jede Eingabezahl wird jeweils die Al-Zahl gesucht, die am nächsten liegt, und der Abstand beider Zahlen ergibt die jeweiligen Kosten, die zum Gesamtergebnis aufaddiert werden.

Zunächst einmal bemerken wir, dass wir eine gültige Lösung finden können, indem wir die durch die Spieler gewählten Zahlen in zehn Gruppen einteilen und dann für jede Gruppe eine Al-Zahl auswählen. Für jede Eingabezahl in einer Gruppe wird am Schluss maximal der Abstand zu der Al-Zahl in ihrer Gruppe bezahlt oder weniger, falls eine Al-Zahl einer fremden Gruppe versehentlich näher liegt. Diese Grundstrategie kann zu einer ausreichend guten Lösung führen, je nachdem, wie die Auswahl der Gruppen und die Auswahl der Al-Zahlen erfolgt.

Wir beschäftigen uns nun im Folgenden mit beiden Lösungskomponenten, also Gruppeneinteilung und Auswahl einer Al-Zahl für die Gruppe:

Auswahl einer Al-Zahl für eine Gruppe

Für eine vorgegebene Menge an Zahlen eine einzige Al-Zahl optimal auszuwählen, ist polynomiell lösbar. Hierzu berechnet man den Median der Zahlen. Der Median ist die Eingabezahl, für die genauso viele Zahlen kleiner sind wie größer. (Ist die Anzahl der Eingabezahlen gerade, muss man sich überlegen, ob es besser ist, eine kleinere Zahl mehr zu haben oder eine größere Zahl; das spielt aber in den vorgegebenen Beispielen der Aufgabe keine Rolle). Wichtig ist, dass man wirklich den Median verwendet, und nicht z. B. den Durchschnitt. Man denke dabei z.B. an eine Gruppe mit sehr vielen kleinen Zahlen und einer großen Zahl: 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 und 50. Hier ist es deutlich teurer, den Durchschnitt zu verwenden (der ca. bei 4,54 liegt, was Kosten von ca. 90 erzeugt), als einfach die Al-Zahl auf die 1 zu legen und nur einmal für die 50 eine längere Strecke zu bezahlen.

Einteilung in Gruppen

Das Problem der Gruppeneinteilung ist deutlich schwieriger; man kann nicht auf einfache Weise im Vorhinein abschätzen, welche Auswahl von Gruppen optimal sein wird. Für eine optimale Gruppeneinteilung benötigt man kompliziertere Ansätze. Hierfür gibt es verschiedene Möglichkeiten wie zum Beispiel:

- Einteilen anhand fixer Gruppenbreite, z. B. alle Zahlen von 1-100, von 101-200, usw.
- Wie eben, aber es werden zumindest leere Intervalle erkannt, die keine Zahl enthalten, und dafür werden dann andere Intervalle weiter aufgeteilt.
- Einteilen nach Anzahlen, d.h. Sortieren der Zahlen und dann die ersten zehn Zahlen in dieser Sortierung, die nächsten zehn Zahlen, usw., als Gruppe wählen.

Partitionierung um Medoiden (PAM)

Statt mit dynamischer Programmierung lässt sich die Aufgabe auch mit PAM (Partitioning Around Medoids)[6] lösen[7]. Hierbei sucht man sich anfangs m zufällige Teilnehmerwerte für Als Zahlen heraus. Anschließend probiert man alle Möglichkeiten aus, diese m Punkte mit einem der $n - m$ unbenutzten Teilnehmerwerte zu tauschen. Es wird immer derjenige Tausch für Al ausgewählt, der ihm am meisten Geld einspart. Dies tun wir solange, bis keine weiteren Vertauschungen mehr möglich sind. Bei PAM handelt es auch um ein Greedy-Verfahren[8], welches in jedem Rechenschritt die lokal beste Lösung sucht. Es ist aber nicht klar, ob die erhaltene Lösung zum Schluss immer die optimale Gesamtlösung für Al darstellt.

Randomisierte Algorithmen

Es lässt sich auch ein randomisiertes Vorgehen anwenden, bei dem die von Al gewählten Zahlen zufällig bestimmt werden. Ein einfaches zufälliges Bestimmen führt allerdings noch nicht zu ausreichend guten Ergebnissen und muss daher noch verbessert werden.

[6] https://de.wikipedia.org/wiki/K-Means-Algorithmus

[7] https://en.wikibooks.org/wiki/Data_Mining_Algorithms_In_R/Clustering/Partitioning_Around_Medoids_-(PAM)

[8] https://de.wikipedia.org/wiki/Greedy-Algorithmus

Aufgrund der Erkenntnis, dass die optimale Lösung immer erreicht werden kann, wenn Zahlen der Teilnehmer gewählt werden, lassen sich allerdings bereits bessere Lösungen erzielen.

Ebenso lässt sich eine Verbesserung dadurch erzielen, indem mehrere Zufallswahlen von Als Zahlen generiert werden, von denen diejenige ausgewählt wird, die die Kosten für Al minimiert. Auch eine schrittweise Verbesserung durch das Verschieben einzelner Werte der Auswahl ist hier denkbar.

3.11 Bewertungskriterien

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

- (1) [-1] **Dokumentation sehr unverständlich bzw. unvollständig**.
- (2) [-1] **Vorgegebenes Dateneingabeformat mangelhaft umgesetzt**:
  Die vorgegebenen Daten sollten aus den Dateien korrekt eingelesen werden und intern geeignet gespeichert werden. Insbesondere sind lineare Datenstrukturen dafür geeignet.
- (3) [-1] **Lösungsverfahren fehlerhaft**:
  Das Lösungsverfahren zur Gewinnoptimierung sollte allgemein genug sein, d.h. nicht auf spezielle Beispiele zugeschnitten. Auch Greedy-Verfahren zur Gewinnoptimierung sind in Ordnung. Der seltene Verlustfall mit einem negativen „Gewinn“ muss nicht extra berücksichtigt und erläutert werden.
- (4) [-1] **Strategie zur Gewinnoptimierung mangelhaft / fehlt**:
  Es sollte eine nachvollziehbare Strategie zur Auswahl von Als Zahlen angewandt und erläutert werden; sie soll zumindest für die vorgegebenen Beispiel zu einem Gewinn für Al führen (Auszahlung $<$ Einsätze). Allzu einfache Strategien sind nicht akzeptabel, wie etwa die Einteilung der Glückszahlen in Gruppen mit Berechnung des Durchschnitts oder gar eine von den Glückszahlen der Teilnehmer unabhängig festgelegte Auswahl der 10 Zahlen für Al.
- (5) [-1] **Verfahren bzw. Implementierung unnötig aufwendig**:
  Nur besonders umständliche Verfahren bzw. Implementierungen führen zu Punktabzug. Die Laufzeiteffizienz wird nicht bewertet.
- (6) [-1] **Programmausgabe ungeeignet**: Die Programmausgabe sollte verständlich sein. Die berechneten Zahlen von Al sowie der Gesamtgewinn und/oder die Gesamtauszahlung sollten ausgegeben werden, auch wenn die letztere Angabe nicht explizit von der Aufgabenstellung verlangt war. Wenn die Angabe des Gesamtgewinns fehlt, ist dies als ein Indiz für ein zweifelhaftes Lösungsverfahren zu werten.
- (7) [-1] **Beispiele fehlerhaft bzw. zu wenige (mind. 2/3)**:
  Es wird die Dokumentation der Ergebnisse von mind. 2 der 3 vorgegebenen Beispiele erwartet.

Aufgabe 4: Schrebergärten

4.1 Einleitung

Die Aufgabe besteht im Grunde daraus, eine gegebene Menge von Rechtecken in ein möglichst kleines umschließendes Rechteck unterzubringen. Dieses Problem kommt in ähnlicher Weise auch in vielen Praxisanwendungen zum Vorschein, neben naheliegenden Problemen wie dem Anordnen von Waren oder dem Kauf von Grundstücken kann es auch für Planungsprobleme benutzt werden, bei dem die Rechtecke beispielsweise Aufgaben repräsentieren, wobei die Breite den Zeitaufwand darstellte und die Höhe den Ressourcenverbrauch.

Auch ist das Problem von Relevanz fürs effiziente Laden von Webseiten: Damit nicht alle Bilder einzeln vom Server abgefragt werden müssen (mit dem einhergehenden Zeitaufwand), werden diese oftmals vor der Datenübertragung auf dem Server zu einem großen Bild zusammengefügt. Dies ist besonders bei Icons und ähnlich kleinen Bildelementen der Fall. Damit sich dies lohnt, darf einerseits das Packen der Bilder nicht zu lange dauern und andererseits darf das neu entstehende Bild nicht viel größer sein als die Summe der Einzelbilder.

In der Forschung ist das Problem auch als „Optimal Rectangle Packing“ bekannt. Es ist NP-schwer und gehört zu den Behälterproblemen[9] („Bin Packing Problems“[10]) der Packungsprobleme („Packing Problems“[11]). Für den interessierten Leser ist ein Paper von Huang und Korf von 2013[12] (mit zusätzlichem Programmcode[13]) zu empfehlen, in dem eine der besten Möglichkeiten zu finden ist, das Problem exakt zu lösen. Im Folgenden werden wir hier aber nur eine einfachere und annähernde Lösung vorstellen, die auf einer anderen Idee basiert.

4.2 Lösungsidee

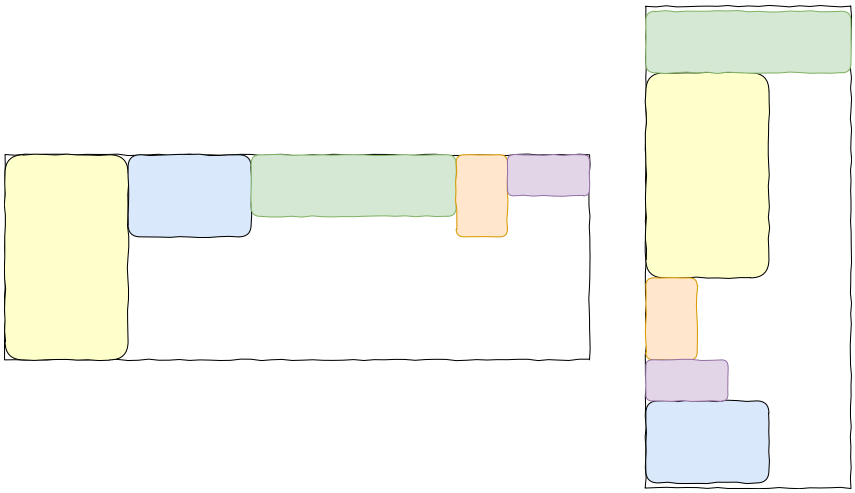


Abbildung 4.1: Einfache senkrechte und waagerechte Anordnung.

[9] https://de.wikipedia.org/wiki/Behälterproblem
[10] https://en.wikipedia.org/wiki/Bin_packing_problem
[11] https://en.wikipedia.org/wiki/Packing_problems
[12] Paper von E. Huang und R. Korf: *Optimal Rectangle Packing: An Absolute Placement Approach*, Journal of Artificial Intelligence Research, 46(1):47-87, 2013. https://jair.org/index.php/jair/article/view/10797/25774
[13] https://github.com/pupitetris/rectpack

Zunächst kann es sich zum besseren Verständnis lohnen, einen Blick auf die einfachen Möglichkeiten lohnen, ein umschließendes Rechteck zu finden. Die einfachste Möglichkeit ist wohl alle Rechtecke nebeneinander oder übereinander, wie beispielhaft geschehen in Abbildung 4.1, anzuordnen. Dies sorgt jedoch in den meisten Fällen für eine sehr hohe Platzverschwendung, ergibt aber bereits die ersten oberen Grenzen für unsere Lösung.

Zum Finden einer guten Lösung müssen nun effizient mehrere mögliche Lösungen durchprobiert werden. Hier wird der von Richard E. Korf entwickelte Algorithmus[14] vorgestellt, abgeändert zu einem annäherndem Verfahren[15] von Matt Perdeck. Dabei werden die möglichen einfassenden Rechtecke von breit nach schmal durchsucht.

1. Sortiere die Rechtecke absteigend nach ihrer Höhe.
2. Beginne mit einem einfassendem Rechteck, das die Höhe des höchsten Rechtecks besitzt und unendliche Breite.
3. Beginne nun die Rechtecke, angefangen mit dem höchsten, zu platzieren: Wähle dabei jeweils den Platz ganz links, der möglich ist; wenn es mehrere Möglichkeiten gibt, wähle den höchsten Platz, siehe Abbildung 4.2.
4. Passe die Breite des umgebenden Rechtecks auf das minimal mögliche an.
5. Konnten alle Rechtecke untergebracht werden?
   a) Ja: Speichere das resultierende umgebende Rechteck, wenn es das kleinste bisherige ist. Verringe die Breite des umgebenden Rechtecks um 1.
   b) Nein: Das geprüfte umgebende Rechteck war zu klein, erhöhe die Höhe um 1.
6. Wenn die Fläche des neuen umgebenden Rechtecks nach der Anpassung nun kleiner ist als die Summe der Flächen aller zu platzierenden Rechtecke, erhöhe die Höhe solange, bis dies nicht mehr der Fall ist. Andernfalls gäbe es keinerlei Möglichkeit, alle Rechtecke unterzubringen.
7. Wenn das neue Rechteck größer ist als das bisher kleinste gefundene, reduziere die Breite solange, bis dies nicht mehr der Fall ist.
8. Wenn das neue umgebende Rechteck nun schlanker ist als das breiteste Rechteck, beende die Suche; es wird kein besseres Rechteck mehr gefunden werden.
9. Gehe zurück zu Schritt 3, um die Suche mit dem neuen umgebenden Rechteck fortzusetzen.

Für eine optimale Lösung müssten in Schritt 5 alle möglichen Anordnungen erprobt werden, wenn das Rechteck nicht passt. Dies würde z.B. mit einer Backtracking- Strategie[16] erfolgen und ist recht zeitaufwändig. Sich auf die sortierte Reihung zu verlassen und mit Hilfe des beschriebenen Greedy-Verfahrens[17] zu prüfen, hat sich in der Praxis als gute und einfache Weise erprobt, solide Lösungen zu erhalten.

Prinzipiell wäre es auch möglich, verschiedene zufällige Reihenfolgen zu erproben, bevor man aufgibt oder andere feste Anordnungen zu testen. Wenn man dies umsetzen würde, wäre es möglich, einen Parameter einzuführen, der die Anzahl der zu testenden Anordnungen angibt und dadurch eine Abwägung zwischen Geschwindigkeit und Güte der Lösungen erlaubt.

[14] Paper von Richard E. Korf: *Optimal Rectangle Packing: Initial Results*, ICAPS 2003 Proceedings. https://www.aaai.org/Papers/ICAPS/2003/ICAPS03-029.pdf

[15] https://www.codeproject.com/Articles/210979/Fast-optimizing-rectangle-packing-algorithm-for-bu

[16] https://de.wikipedia.org/wiki/Backtracking

[17] https://de.wikipedia.org/wiki/Greedy-Algorithmus

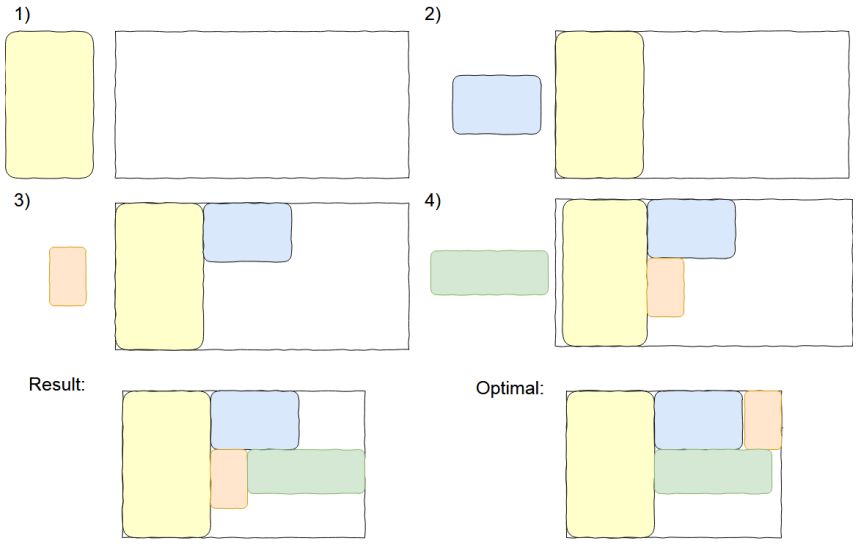


Abbildung 4.2: Beispielhafte Ausführung von Schritt 3 für vier zu platzierende Rechtecke. Jeweils das momentan einzufügende Rechteck und der Rand des umgebenden Rechtecks vor dem Einfügen sind dargestellt.

4.3 Platzierung der Rechtecke

Um wie in Schritt 3 beschrieben den Platz ganz links zu finden, gibt es mehrere Möglichkeiten. Der naive Ansatz besteht darin, in einem zweidimensionalen Array alle belegte Plätze zu markieren und bei der Platzierung dann von oben nach unten und von links nach rechts zu iterieren. Dies ist jedoch nicht wirklich effizient.

Eine Möglichkeit, dies zu verbessern, ist eine Datenstruktur, die ähnlich zu einer der oben genannten Lösungsideen Felder verwaltet, die belegt oder frei sein können. Dabei assoziiert man jede Spalte mit einer Breite und jedes Feld mit einer Höhe, und es werden nur so viele Felder erzeugt, wie benötigt werden.

Man beginnt dabei mit einem einzigen Feld, dessen Höhe und Breite dem zu testenden einfassenden Rechteck entspricht. Rechtecke werden als Paar von Zahlen, die Breite und Höhe entsprechen, dargestellt. Wird nun ein Rechteck hinzugefügt, ist dies einfach in dieses freie Feld möglich. Dabei entsteht jedoch ein Feld, das sowohl belegt, als auch unbelegt ist. Daher wird dieses Feld geteilt: horizontal auf der Höhe des Rechtecks und vertikal in der Breite. Das Ergebnis sind vier Felder.

Soll nun ein neues Rechteck hinzugefügt werden, wird durch diese Felder entsprechend von links nach rechts und von oben nach unten iteriert, um einen möglichen Platz zu finden. Wird ein solcher gefunden, kann eingefügt werden, wobei eventuell wiederum Felder geteilt werden müssen.

Am besten wird die Idee mit einem kurzen Beispiel verdeutlicht (siehe Abbildung 4.3). Man beginnt mit einem einzigen leeren Feld, dessen Abmessungen denen des zu prüfenden einfassenden Rechtecks entsprechen. Wird nun ein neues Rechteck hinzugefügt (1), ist die Position für dieses leicht zu finden, es entspricht der oberen linken Ecke des einzigen Feldes. Beim Einfügen tritt nun aber eine Situation ein, bei der ein Feld sowohl belegt als auch frei ist. Daher wird das Feld nun in vier neue Felder gespalten (2).

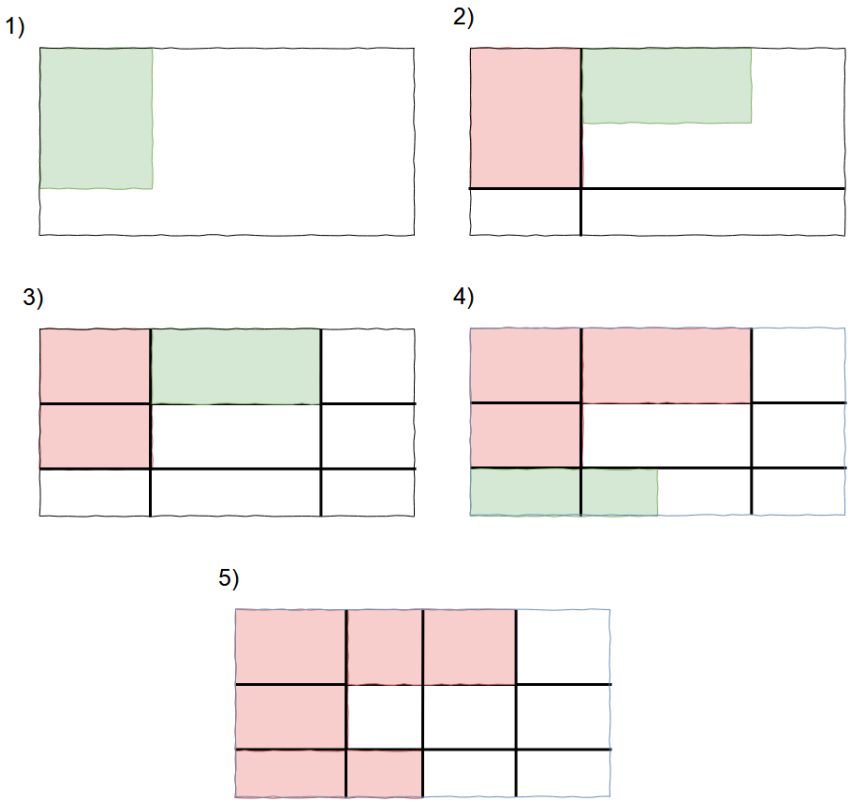


Abbildung 4.3: Beispiel für die beschrieben Datenstruktur aus Feldern (neu eingefügte Rechtecke in grün, belegte Felder in rot).

Wenn nun ein neues Rechteck eingefügt werden soll, wird zunächst durch jede Spalte, d.h. von links nach rechts und von oben nach unten iteriert. Im Beispiel des neu eingefügten Rechtecks war das Feld unter dem bereits eingefügten Rechteck nicht hoch genug, daher wird es in das obere rechte Feld eingefügt. Hier kommt es wieder vor, dass Felder geteilt werden müssen. In (4) ist das untere Feld hoch genug, so dass noch geprüft werden muss, ob zusammen mit nebeneinanderliegenden Felder genug Breite vorhanden ist.

Verbesserungen

Statt bei Schritt 5b) die Höhe nur um 1 zu erhöhen, kann man sich bei der vorgestellten Heuristik überlegen, dass eine größere Erhöhung möglich ist. Es muss nämlich entweder um die Höhe des Rechteckes erhöht werden, das nicht platziert werden konnte oder aber soweit, dass die bisher platzierten Rechtecke anders platziert werden können. Die erste Zahl ist leicht zu bekommen, für den zweiten Wert ist kurzes Nachdenken erforderlich. Aber auch diese Zahl ist kein Problem, man prüft für jedes Rechteck, das nicht in einem Feld platziert werden konnte, wie viel Höhe mehr erforderlich wäre, damit eine Platzierung gelingt und merkt sich das Minimum. Dann wird in Schritt 5a) statt um 1 das Rechteck um das Minimum der zwei vorgestellten Werte erhöht.

Eine weitere Überlegung wäre, einen Parameter einzuführen, bei dem man zufällige Reihenfolgen statt der sortierten Reihenfolge testet. Durch das Testen einer einzigen Reihenfolge ist das Programm sehr schnell und kommt mit einer sehr großen Anzahl an Schrebergärten zurecht, jedoch geht dies zu Lasten der Güte der Lösung. Mit Hilfe eines solchen Parameters könnte man Qualität und Geschwindigkeit manuell gegeneinander abwägen.

4.4 Alternative Lösungsideen

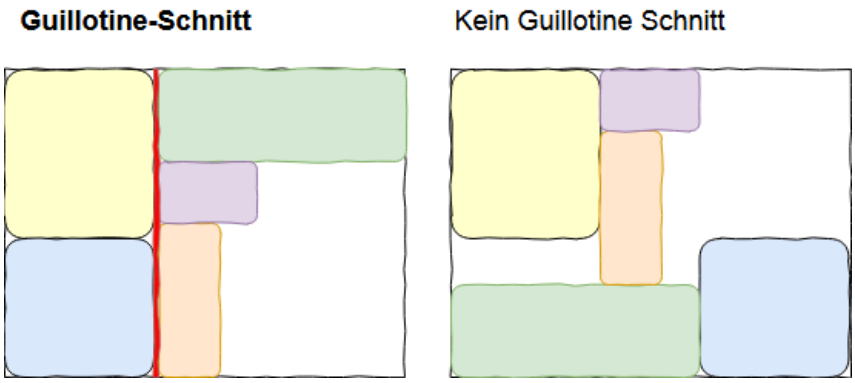


Abbildung 4.4: Beispiel für Anordnungen mit und ohne Guillotonie-Schnitt.

Die Aufgabenstellung erlaubt zur Lösung viele verschiedene Lösungswege und Heuristiken. Meist werden verschiedene Lösungen erprobt und vergleichend evaluiert. Dies kann auf unterschiedliche Arten geschehen, im akademischen Bereich ist das Problem ergiebig analysiert. Verbesserungen zum oben genannten Verfahren werden beispielsweise in einem weiteren Paper[18] von Richard E. Korf vorgestellt.

Eine Übersicht über verschiedene Verfahren bietet u.a. ein Paper[19] von Jukka Jylänki mit entsprechendem Programmcode[20] und visueller Darstellung der Algorithmen[21].

Weitere populäre Ansätze sind beispielsweise Lösungen mit sogenannten Guillotine-Schnitten zu betrachten, d.h. Lösungen, bei denen man das umfassende Rechteck sozusagen zerschneiden kann, ohne dabei die enthaltenen Rechtecke zu zerschneiden. Veranschaulicht ist diese Lösungsidee in Abb. 4.4.

Außerdem sind in der Tabelle 4.5 die optimalen Lösungen für das Packen von aufsteigenden Quadraten enthalten; an diesen kann sich orientiert werden, um die Lösungsgüte zu einzuschätzen. Dabei sind exakte Lösungen bis Größe 15 in wenigen Sekunden möglich; Heuristiken sollten deutlich mehr schaffen.

4.5 Beispiele

Im Folgenden werden die optimalen und die vom Programm errechneten Lösungen an Hand von einigen Beispielen verglichen, siehe Abbildung 4.6.

Wie bereits erwähnt, könnte die Lösungsgüte auf Kosten der Performance verbessert werden, indem man mehrere Permutationen bei der Platzierung prüft, statt nur die sortierte Reihenfolge der Rechtecke durchzurechnen.

[18] Paper von Richard E. Korf: *Optimal Rectangle Packing: New Results*, ICAPS 2004 Proceedings. https://www.aaai.org/Papers/ICAPS/2004/ICAPS04-019.pdf

[19] Paper von Jukka Jylänki: *A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing*, 2010. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.695.2918

[20] https://github.com/juj/RectangleBinPack

[21] http://www.binpacking.4fan.cz/

Size N	Optimal Solution	Waste Percent
1	1×1	0%
2	2×3	16.67%
3	3×5	6.67%
4	5×7	14.29%
5	5×12	8.33%
6	9×11	8.08%
7	7×22	9.09%
7	11×14	9.09%
8	14×15	2.86%
9	15×20	5.00%
10	15×27	4.94%
11	19×27	1.36%
12	23×29	2.55%
13	22×38	2.03%
14	23×45	1.93%
15	23×55	1.98%
16	27×56	1.06%
16	28×54	1.06%
17	39×46	0.50%
18	31×69	1.40%
19	47×53	0.84%
20	34×85	0.69%
21	38×85	0.99%
22	39×98	0.71%
23	64×68	0.64%
24	56×88	0.57%
25	43×129	0.40%

Abbildung 4.5: Tabelle für die optimale Lösungen der Anordnung von aufsteigend größeren Quadraten, d.h. 1x1,2x2,... *nxn*, entnommen aus Table 1 des Papers von Richard E. Korf: *Optimal Rectangle Packing: New Results*, ICAPS 2004 Proceedings. https://www.aaai.org/Papers/ICAPS/2004/ICAPS04-019.pdf

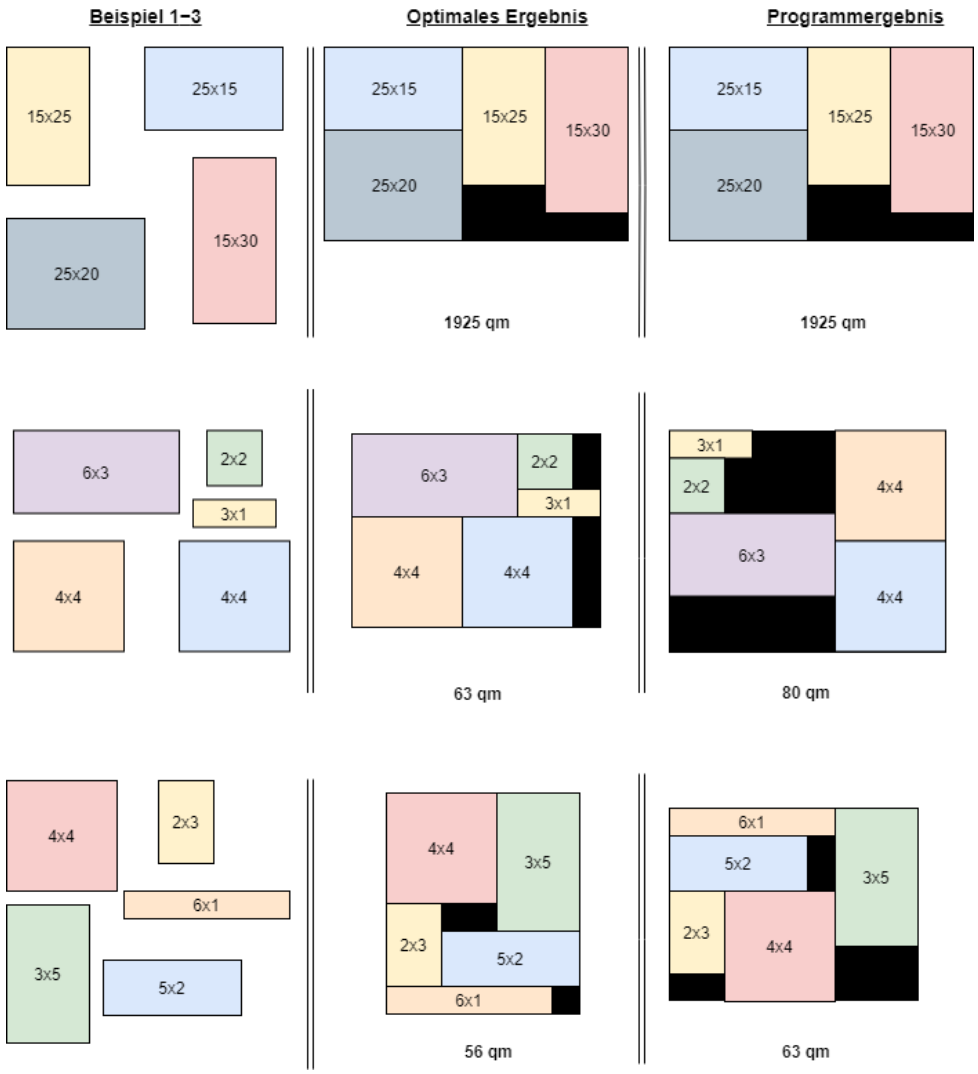


Abbildung 4.6: Beispiele samt optimaler und vom Programm errechneter Lösung.

4.6 Bewertungskriterien

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

- (1) [-1] **Dokumentation sehr unverständlich bzw. unvollständig**.
- (2) [-1] **Vorgegebenes Dateneingabeformat mangelhaft umgesetzt**:
  Die vorgegebenen Daten sollten vom Programm intern geeignet gespeichert werden. Insbesondere sind Arrays dafür geeignet. Statt aus einer Datei oder mittels einer Eingabezeile können die Rechteckdaten auch z. B als Konstanten im Programmcode enthalten sein.
- (3) [-1] **Lösungsverfahren fehlerhaft**:
  Das Lösungsverfahren sollte korrekt sein und allgemein genug sein, d.h. nicht auf spezielle Beispiele zugeschnitten. Auch sollte es nicht auf bis zu vier Rechtecke oder auf eine andere Zahl von Rechtecken festgelegt sein. Die Geometrie der Rechtecke sollte korrekt umgesetzt sein (z.B. Beachten von Schnittpunkten und Überlappungen). Eine Drehung der Rechtecke war durch die Aufgabenstellung eigentlich ausgeschlossen, aber da sie die Aufgabe schwieriger macht, gibt es dafür keinen Punktabzug.

- (4) [-1] **Strategie zur Optimierung mangelhaft / fehlt**:
  Grundsätzlich gibt es bei diesem Problem etliche Lösungsmöglichkeiten, weshalb beim Lösungsansatz viel Spielraum gelassen werden sollte. Auf jeden Fall sollte der Flächeninhalt des die einzelnen Schrebergärten umgebenden Rechtecks die maßgebliche Größe für die Optimierung sein. Auch Greedy-Verfahren zur Optimierung sind in Ordnung. Wichtig ist jedoch, dass klar dokumentiert ist, ob der gewählte Lösungsansatz optimale oder approximative Lösungen liefert. Gedanken und Angaben zur konkret möglichen Flächenersparnis wären zwar schön (z.B. theoretische Überlegungen zum Optimum versus Approximation oder ein Vergleich der Ergebnisse unterschiedlicher Rechenverfahren anhand von Beispielen), sind aber von der Aufgabenstellung nicht gefordert und können daher im Falle einer fehlerhaften Beschreibung nicht zu Punktabzug führen.
- (5) [-1] **Verfahren bzw. Implementierung unnötig aufwendig**:
  Nur besonders umständliche Verfahren bzw. Implementierungen führen zu Punktabzug. Die Laufzeiteffizienz wird nicht bewertet.
- (6) [-1] **Programmausgabe ungeeignet**:
  Die Programmausgabe sollte mittels einer visuellen Darstellung erfolgen und nachvollziehbar sein. Grafische Darstellungen sind schön, aber auch Visualisierungen mit Textzeichen sind akzeptabel. Wichtig ist, dass die eingegebenen Rechtecke erkennbar sind.
- (7) [-1] **Beispiele fehlerhaft bzw. zu wenige (mind. 2/3)**:
  Es wird die Dokumentation und visuelle Darstellung der Ergebnisse von mind. 2 der 3 vorgegebenen Beispiele erwartet.

Aufgabe 5: Widerstand

5.1 Lösungsidee

Modellierung einer Schaltung

Nehmen wir für einen Augenblick an, wir hätten bereits ausgewählt, welche Widerstände wir nutzen wollen und wie wir diese anordnen. Wie können wir diese Schaltung nun in unserem Programm darstellen und den Widerstandswert berechnen?

Wie in der Aufgabenstellung beschrieben, kann jede Schaltung in mehreren Schritten aufgebaut werden, wobei in jedem Schritt zwei Teilschaltungen verbunden werden. Für diese Verbindung gibt es zwei Möglichkeiten: Man kann die Widerstände entweder parallel oder in Reihe schalten.

Diese Struktur kann gut durch einen binären Baum dargestellt werden (Abbildung 5.1). Jedes Blatt stellt einen Widerstand aus der Grabbelkiste und jeder innere Knoten eine Verbindung dar. Der Widerstandswert jeder Teilschaltung lässt sich leicht aus den Widerstandswerten ihrer Kinder berechnen.

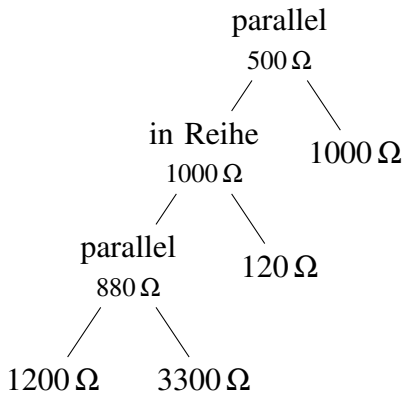


Abbildung 5.1: Darstellung des Beispiels aus der Aufgabenstellung als Baum.

Struktur des Schaltungsbaums

Da die Schaltung aus maximal vier Widerständen bestehen soll, hat auch der Schaltungsbaum höchstens vier Blätter. Es gibt nur sehr wenige Möglichkeiten, einen Binärbaum mit bis zu vier Blättern zu konstruieren.

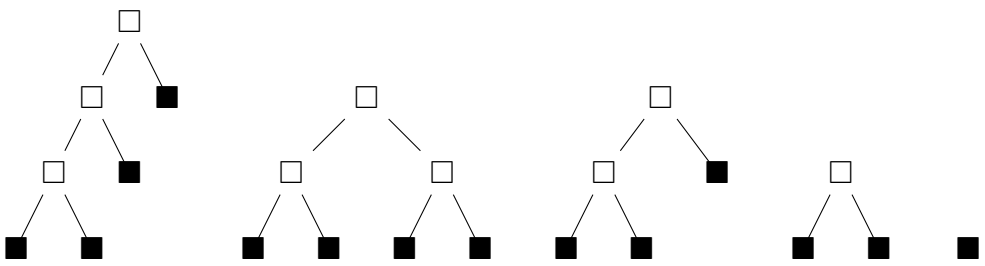


Abbildung 5.2: Alle möglichen Arten von Schaltungsbäumen.

Bäume, die durch Vertauschen der Kinder eines Knotens aus einem anderen Baum entstehen, brauchen wir nicht zu berücksichtigen, da es irrelevant ist, welcher von zwei Widerständen in einer Reihenschaltung *vorne* oder *hinten* bzw. in einer Parallelschaltung *oben* oder *unten* ist. Das ergibt sich daraus, dass die Formeln für die Berechnung des resultierenden Widerstands kommutativ sind, also $a+b=b+a$ und $\frac{1}{\frac{1}{a}+\frac{1}{b}} = \frac{1}{\frac{1}{b}+\frac{1}{a}}$ für zwei Widerstandswerte a und b.

Ein Sonderfall lässt sich nicht als Baum darstellen: die Schaltung, die nur aus einem Kabel ohne Widerstand besteht, mit einem Widerstandswert von $0\,\Omega$. Dieser lässt sich aber einfach getrennt behandeln.

Um alle möglichen Schaltungen auszuprobieren, müssen wir nur für die obigen *Baumarten* für jedes weiße Kästchen "parallel" und "in Reihe" und für jedes schwarze Kästchen alle möglichen Widerstände ausprobieren (Abbildung 5.2). Bei den Widerständen müssen wir darauf achten, dass sie nicht innerhalb einer Schaltung wiederverwendet werden können. In der Schaltung dürfen zum Beispiel nur dann zwei $150\,\Omega$-Widerstände vorkommen, wenn auch die Eingabe diesen zweimal enthält. Diejenigen Möglichkeiten, die duplizierte Widerstände enthalten, filtern wir also aus.

5.2 Optimierungen

Binäre Suche

Eine Möglichkeit zur Optimierung besteht in der Auswahl der Widerstände, die benutzt werden. Wir wählen erst mal nur drei statt vier Widerstände, sowie einen Platzhalter, und konstruieren daraus Bäume. Für jeden Baum können wir nun berechnen, was der optimale Widerstandswert für den Platzhalter wäre. Man sucht also ein x, sodass der Widerstand der Schaltung insgesamt R wird.

Im nächsten Schritt sucht man die beiden Widerstände, die am nächsten an x herankommen (ein Wert $\geq x$ und einer $\leq x$) und setzt diese an Stelle des Platzhalters ein. Eine dieser beiden Schaltungen wird das beste Ergebnis liefern, das mit den gewählten drei Widerständen und der Baumstruktur möglich ist. Mann muss also nur zwei Möglichkeiten für den vierten Widerstand ausprobieren. Dies funktioniert nur, da der Widerstandswert der Schaltung abhängig von x streng monoton steigend ist.

Wie berechnet man den Wert x? Am besten rekursiv: Sei v der Widerstand der Teilschaltung ohne Platzhalter. Wenn der aktuellen Baumknoten eine Reihenschaltung darstellt, muss die Teilschaltung mit Platzhalter den Widerstandswert $R-v$ haben, um insgesamt R zu erreichen, bei einer Parallelschaltung stattdessen $\frac{1}{\frac{1}{R}-\frac{1}{v}}$.

5.3 Laufzeit

Sei n die Anzahl der Widerstände in der Grabbelkiste.

Der unoptimierte Algorithmus probiert für (fast) jede Auswahl von vier Widerständen aus dieser Kiste jeweils bis zu 2^3 Möglichkeiten (Zuweisungen von "parallel" und "in Reihe") für die fünf möglichen Baumstrukturen aus. Es gibt n^4 Möglichkeiten, vier Widerstände aus der Kiste auszuwählen. Damit kommen wir auf bis zu $2^3 \cdot 5 \cdot n^4 = 40n^4$ ausprobierte Bäume. Die Laufzeit des Algorithmus liegt also in $\mathcal{O}(n^4)$.

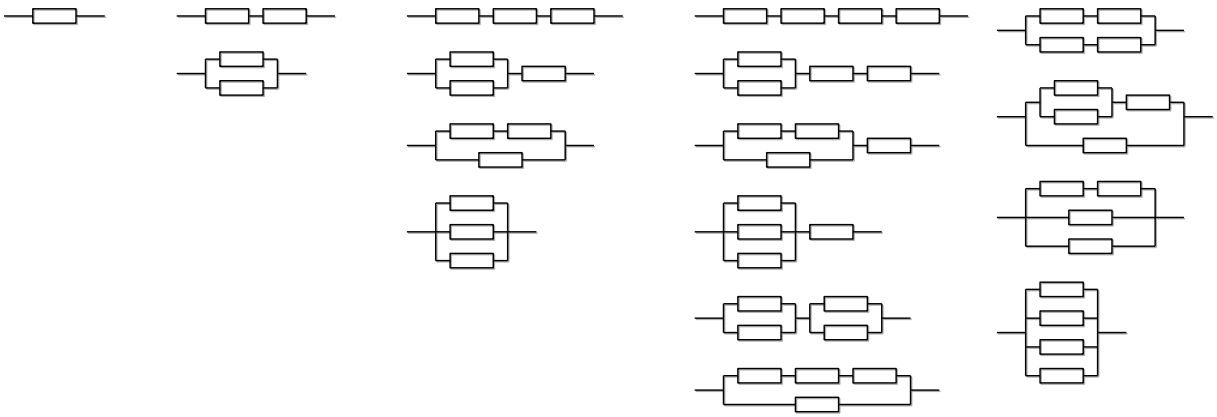
Der Algorithmus mit binärer Suche muss nur drei Widerstände aus der Kiste auswählen. Die binäre Suche zum Finden des vierten Widerstands läuft jeweils in $\mathcal{O}(\log_2(n))$. Dass wir am Anfang die Widerstände nach Wert sortieren müssen (typische Laufzeit in $\mathcal{O}(n\log_2(n))$), fällt nicht weiter ins Gewicht. Insgesamt kommen wir auf eine Laufzeit von $\mathcal{O}(n^3\log_2(n))$.

Das ist bereits ein deutlicher Unterschied, wie man an den folgenden Laufzeitmessungen sieht:

Eingabe	n = 23 Beispieleingabe	n = 50 Zufallszahlen	n = 100 Zufallszahlen
$\mathcal{O}(n^4)$-Algorithmus	1.26s	27.24s	460.25s
$\mathcal{O}(n^3\log_2(n))$-Algorithmus	0.28s	2.69s	22.41s

5.4 Alternative Lösungsideen

Da die Anzahl der zu betrachtenden Schaltungen der bis zu $k = 4$ Widerstände noch überschaubar ist, können alternativ auch alle möglichen Schaltungen bis $k = 4$ betrachtet werden. Dabei kommen für $k = 1,2,3,4$ die folgenden 1, 2, 4 bzw. 10 Schaltungen in Betracht:



Andererseits kann man auch vollkommen auf das Verwenden vorgegebener Schaltungsgraphen oder -bäume verzichten und die Schaltung komplett rekursiv durch die Parallel- und Reihenschaltung von vorhandenen und bereits zusammengebauten Widerständen aufbauen. Dabei müssen aber die Abbruchkriterien geeignet gewählt werden.

5.5 Beispiele

Für die vorgegebenen Beispiele berechnete das Programm die folgenden Ergebnisse:

R = 500:

```
k = 1: (Widerstand 470 Ω)

  --| 470 |--

k = 2: (Widerstand 500.380228 Ω)

   |--| 4700 |--|
  -|            |-
   |--| 560 |---|

k = 3: (Widerstand 500 Ω)

  --| 220 |----| 100 |----| 180 |--

k = 4: (Widerstand 500 Ω)

  --| 220 |----| 100 |----| 180 |--
```

R = 140:

```
k = 1: (Widerstand 150 Ω)

  --| 150 |--

k = 2: (Widerstand 140.425532 Ω)

   |--| 2200 |--|
  -|            |-
   |--| 150 |---|

k = 3: (Widerstand 139.949431 Ω)

     |--| 2700 |--|
   |-|            |-|
   | |--| 820 |---| |
  -|                 |-
   |--| 180 |-------|

k = 4: (Widerstand 140 Ω)

   |--| 1800 |----| 120 |----| 180 |--|
  -|                                    |-
   |--| 150 |-------------------------|
```

R = 314:

```
k = 1: (Widerstand 330 Ω)

  --| 330 |--

k = 2: (Widerstand 314.726508 Ω)

   |--| 6800 |--|
  -|            |-
   |--| 330 |---|

k = 3: (Widerstand 314.055637 Ω)

   |--| 4700 |----| 1800 |--|
  -|                        |-
   |--| 330 |---------------|

k = 4: (Widerstand 313.999343 Ω)

     |--| 1200 |--|
   |-|            |---| 120 |--|
   | |--| 470 |---|            |
  -|                           |-
   |--| 1000 |-----------------|
```

R = 315:

```
k = 1: (Widerstand 330 Ω)

  --| 330 |--

k = 2: (Widerstand 314.726508 Ω)

   |--| 6800 |--|
  -|            |-
   |--| 330 |---|

k = 3: (Widerstand 315 Ω)

   |--| 330 |----| 390 |--|
  -|                      |-
   |--| 560 |-------------|

k = 4: (Widerstand 315 Ω)

   |--| 330 |----| 390 |--|
  -|                      |-
   |--| 560 |-------------|
```

R = 1620:

```
k = 1: (Widerstand 1500 Ω)

  --| 1500 |--

k = 2: (Widerstand 1620 Ω)

  --| 1500 |----| 120 |--

k = 3: (Widerstand 1620 Ω)

  --| 1500 |----| 120 |--

k = 4: (Widerstand 1620 Ω)

  --| 1500 |----| 120 |--
```

R = 2719:

```
k = 1: (Widerstand 2700 Ω)

  --| 2700 |--

k = 2: (Widerstand 2700 Ω)

  --| 2700 |--

k = 3: (Widerstand 2720 Ω)

  --| 1800 |----| 100 |----| 820 |--

k = 4: (Widerstand 2719 Ω)

   |--| 180 |--|
  -|           |---| 1800 |----| 820 |--
   |--| 220 |--|
```

R = 4242:

```
k = 1: (Widerstand 3900 Ω)

  --| 3900 |--

k = 2: (Widerstand 4230 Ω)

  --| 3900 |----| 330 |--

k = 3: (Widerstand 4240.7767 Ω)

   |--| 390 |---|
  -|             |---| 3900 |--
   |--| 2700 |--|

k = 4: (Widerstand 4242 Ω)

   |--| 120 |--|
  -|           |---| 3900 |----| 270 |--
   |--| 180 |--|
```

5.6 Bewertungskriterien

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

- (1) [-1] **Dokumentation sehr unverständlich bzw. unvollständig**.
- (2) [-1] **Vorgegebenes Dateneingabeformat mangelhaft umgesetzt**:
  Die vorgegebenen Widerstände sollten aus der Datei korrekt eingelesen und geeignet abgespeichert werden. Es ist akzeptabel, wenn die Eingabe nur ganze Zahlen zulässt. Das Programm muss intern aber mit Gleitkommazahlen oder Brüchen rechnen, um für alle Schaltungen den korrekten Widerstandswert ermitteln zu können.
- (3) [-1] **Lösungsverfahren fehlerhaft**:
  Das Lösungsverfahren sollte korrekt sein. Die einzelnen Widerstände sollten zwar nicht öfter als in der Grabbelkiste vorhanden in einer Schaltung verwendet werden, andernfalls wird aber auch kein Punkt abgezogen. Die Minimierung der Widerstandsdifferenz genügt, andere physikalisch sinnvolle Metriken werden nicht gefordert. Eine Berücksichtigung des Sonderfalls von 0 Widerständen bzw. $0\,\Omega$ bleibt unbewertet, auch wenn er dokumentiert wurde.
- (4) [-1] **Strategie für Schaltungsentwurf mangelhaft / fehlt**:
  Das Verfahren sollte immer das optimale Ergebnis zu berechnen versuchen und keinen heuristischen Ansatz folgen. Aufgrund der geringen Anzahl von Schaltungen ist eine Brute-Force-Strategie erlaubt. Eine naive Festverdrahtung aller möglichen Schaltungen durch Programmcode (statt der Verwendung einer Datenstruktur) führt zu Punktabzug, wenn sie mit einem erheblichen Anwachsen des Programmcodes einhergeht (d.h. nicht

nur eine deklarative Codezeile für jede mögliche Schaltung). Alle zehn möglichen Schaltungen, d. h. alle möglichen Reihen- und Parallelschaltungen der bis zu vier Widerstände sollten von der Strategie berücksichtigt werden.

- (5) [-1] **Verfahren bzw. Implementierung unnötig aufwendig**:
  Nur besonders umständliche Verfahren bzw. Implementierungen führen zu Punktabzug. Überlegungen zur Laufzeit, z. B in Abhängigkeit von der Anzahl verfügbarer Widerstände, werden nicht bewertet.
- (6) [-1] **Programmausgabe ungeeignet**:
  Die Programmausgabe sollte nachvollziehbar sein. Grafische Ausgaben sind schön, aber eine textuelle Ausgabe, welche die Struktur des Bauplans nachvollziehbar angibt, genügt. Sowohl der Bauplan der jeweils berechneten Schaltung mit ihren einzelnen Widerstandswerten als auch ihr Gesamtwiderstandswert R sollten ausgegeben werden. Sollte der geforderte Gesamtwiderstand R bereits mit weniger als 4 Widerständen in einer Schaltung möglich sein, so genügt ein Hinweis darauf. Die alleinige Angabe von Berechnungsformeln für den Gesamtwiderstand R ohne einen konkreten Bauplan führt zu Punktabzug.
- (7) [-1] **Beispiele fehlerhaft bzw. zu wenige**:
  Es wird die Dokumentation der Ergebnisse von mind. 2 aussagekräftigen Beispielen mit korrekten Ergebnisse erwartet, die die Güte des Programms sinnvoll demonstrieren. Wurden für k von 1 bis 4 Schaltungen berechnet, aber nur die beste Schaltung, d.h. die mit der geringster Widerstandsdifferenz zum gewünschten Gesamtwiderstand R, ausgegeben, wird kein Punkt abgezogen. Eigentlich war für jedes k von 1 bis 4 (bzw. bis der Wert R erreicht ist) die Ausgabe einer geeigneten Schaltung als Ergebnis erwartet worden, aber einige fanden die Aufgabenstellung in dieser Hinsicht nicht klar genug.

Anhang: Perlen der Informatik aus den Einsendungen

Allgemeines

- „Variabel", „Interger", „Algorytthmu", „Brutforce", „TennieGram", „ich würfel" *– (Verb?)*
- „optimalsten Nährungsverfahren", „expotentiell"
- „Jedoch kann ich sagen, dass O(80) = 180 ist. . . "
- „Das Programm wurde in Java implementiert, ist jedoch nicht strikt objektorientiert."
- „Python brilliert durch seine built-in functions."
- „Dazu braucht man ein flexibles System aus Schleifen und Bedingungen."
- „Da wir Zuversicht in dieses System haben, empfinden wir den Gebrauch von Pseudocode als obsolet."

Junioraufgabe 1: Auf und Ab

- „Da man keine Null oder eine kleinere Zahl würfeln kann, gibt das Programm "Netter Versuch, aber das Programm ist perfekt!!!" aus."
- „Durch die Verschiedenen Sicherungen verhindert das Programm viele Fehler und ist vor falschr Bedinung geschützt."
- „Junior-Junior-Aufgabe 1" *– (Tippfehler in der Kopfzeile.)*
- „[...] ein Programe entwickeln, welches mithilfe eines speziellen Logorithmus herausfindet, welcher Fall funktioniert [...]"
- „Würde man nur Einsen würfeln, benötigt man 26 Würfe, was mit einer Wahrscheinlichkeit von 1 / 170581728179578208256 eintritt."

Junioraufgabe 2: Baywatch

- „Information: Das Programm funktioniert nur unter der Annahme, dass sich der Papagei nicht beim Einprägen der Insel vertan hat."
- Programmausgabe: „Es gibt mehrere eindeutige Lösungen."
- „Papageinliste"
- „Zu dieser Aufgabe gibt es eigentlich nicht viel zu sagen, jedoch gibt es etwas: Meine Idee hinter der Lösung war, dass eigentlich nur ein 'print' Befehl die Lösung hervorbringen kann."

Aufgabe 1: Superstar

- „[Es] wird in einer weiteren for-Schleife der Boolean-Speicher ausgewertet."
- „Das Programm anfangs geht davon aus, dass jedes Gruppenmitglied ein potenzielles Gruppenmitglied ist."

- Ein einziger Satz als Lösungsidee: „Setze genau (!) das um, was in der Aufgabenstellung steht!"
- Ein einziger Satz als Umsetzung: „Die Lösungsidee wird in Whitespace implementiert."
- „[...] Es müssen zwei ganz spezifische Kriterien erfüllt sein, damit ein Mitglied ein Superman ist." *– (Gefolgt von etlichen weiteren Vorkommen des Wortes Superman statt Superstar.)*
- „[...] wird überprüft, ob ein Superstar existiert und sollte dieser zustimmen, wird dieser namentlich ausgegeben."
- „Die Lösungsidee unseres Problems besteht aus einem einfachen Ursache-Wirkungsprinzip."

Aufgabe 2: Twist

- „Twist-Algorythmus"
- „[...] und die Schleife kann gebreakt werden." *– (Breakdance statt Twist.)*
- Programmausgabe: „Hello and Welcome to the Twister"
- „Die Umsetzung verfolgt [...] ähnlich zum ersten Teilprogramm".
- Enttwist-Ergebnis: „Der Aufbruch von Franz null" *– (Statt Franz Kafka.)*
- „Falls false, dann wird die Schleife beendet, um unnötige Laufzeit zu verursachen."
- „Goethe scheint bis auf eine Ausnahme ausschließlich Wörter aus der Beispielwörterliste zu verwenden."
- „Innerhalb der [Enttwister-]Methode wird das Wort zufällig angeordnet und dann mit Firefox Selenium an den Duden geschickt, um das Wort zu überprüfen."
- „Wörterdictionary"
- „Jeder einzelne Buchstabe ist ein bestimmter Ordner, auf den zugegriffen wird. So wird bei dem Wort "Hexenmeister" auf den Pfad "upper/H/r/e/e/e/e/i/m/n/s/t/x/word.txt" zugegriffen, in welchem dann das richtige Wort liegt." *– (Leider stürzte der Computer beim Testen des Programms mit einem großen Beispiel ab.)*
- „Meine Idee war es die Wörter, die mehr als 4 Zeilen besitzen, [...]"
- „Aufgrund von Fehlern im Quellcode, die wir nicht finden konnten, konnten wir das Programm leider nicht kompilieren und somit nicht auf die Beispiele anwenden."
- „Zum Schluss muss das neue Wort noch gedruckt werden."
- „Sie nimmt eine Liste als einzigen Parameter und sortiert sie in zufälliger Reihenfolge."
- „[. . .] eigentlich könnte man das finden von Wörtern zu eine Funktion definieren, aber ich hatte das Gefühl es nicht völlig nötig zu haben."
- Ausgabe-Warnungsmeldung: „Ambiguous word spotted! Program ran into a situation in which a word could be interpreted in multiple ways, it had to decide between following words: ein, ein. The program decided to replace "ein" with "ein"."
- „Als erstes nehmen wir den eingegebenen Text und teilen ihn an jeder Stelle wo eine Leertaste ist auf."
- „Hierbei tritt das Problem auf: Datenmenge."

- „Diesen Algorithmus würde ich nicht kostenlos zur Verfügung stellen."
- „[Das getwistete Wort] kann identisch mid dem ursprünglichen Wort sind [sic]. Unter kryptografischen Gesichtspunkten ist dies jedoch ein Vorteil, da sich eine Kombination so nicht direkt ausschließen lässt - wie die Deutschen mit Enigma erfahren haben."

Aufgabe 3: Voll daneben

- „[...] Doch es ist auffällig, dass es keinen Gewinn, sondern einen Verlust gibt. Um dieses Problem zu lösen, kann Al einfach den Eintrittspreis erhöhen."
- „[...] lässt ein Brute-To-Force-Verfahren scheitern."
- „Danach wird eine unendliche Schleife so lange wiederholt, bis die Anzahl der Durchläufe 100.000 ist."
- Methodenname: „KomplizierteAuswahl"
- „[...] das Programm [kann] mit allen Zahlen innerhalb des Integerbereichs arbeiten, dazu zählen alle Zahlen zwischen 2^{31} und 2^{31}-1."
- „do { ... } while (false);"
- „NP-komplett"
- „Dadurch, dass der Befehl zum Sortieren von Arrays (zumindest auf meinem PC) nicht ordnungsgemäß funktioniert, werden die Ergebnisse hier falsch angezeigt, was aber auf das Setup dieses PCs zurückzuführen sein könnte."
- „Dabei wird nicht angestrebt, die beste Lösung zu ermitteln, da die Teilnehmer schließlich bei einem Glücksspiel auch gewinnen können. Außerdem sollen die Teilnehmer auch belohnt werden, wenn sie ihre Zahlen zusammen intelligent auswählen, [...]"
- „beilspiel1.txt"
- „Wie Mann sieht, sind dies nicht viele [...]"
- „Leider gibt es 9,6 x 1029 Zahlenkombinationen."
- „Die Umsetzung verlief nicht allzu erfolgreich, wenn doch die besten Lösungsansätze gegeben waren. Gründe hierfür waren vor allem fehlende Vorstellungen einer exakten Programmierung."
- Die Lösungsidee wird in Python implementiert, da dies eine sehr einfache Sprache ist, die nicht unnötigerweise den Code verlängert, wie z. B. Java."
- „Um die Zahlen von Al Chambro [sic] zu ermitteln, werden verschiedene Methoden ausprobiert und die effektivste Methode legt anschließend die Zahlen fest."
- „[Es] wird der Mittelwert [...] berechnet. Daraufhin werden Zufallszahlen erzeugt, deren Spielraum stark eingeschränkt wurde, sodass nur der zuvor berechnete Mittelwert entstehen kann. So entstehen nacheinander zehn Zufallszahlen."

Aufgabe 4: Schrebergärten

- "NP-komplett"

- „Die Idee ist, dass die verschieden großen Gärten in verschiedenen Reihenfolgen, an unterschiedlichexn Positionen nach dem Tetris-Prinzip aneinander gestapelt werden. Um von Anfang an die Anzahl an Möglichkeiten zu begrenzen werden die Gärten nicht an beliebigen x-Werten fallen gelassen sondern nur auf den Höhen von den Kanten der schon gesetzten Gärten und eben der Seitenkante x = 0." – *(Mehr Kommas würden der Verständlichkeit des Textes helfen.)*

Aufgabe 5: Widerstand

- „Angefragter Widerstand: 10 Ohmen"
- „[...] Dieser Fehler wurde aber auch nach 10 Stunden Computer Anschreien nicht gefunden."
- „Um die Schaltung graphisch aufs Terminal zu bringen, musste eine Art Programmiersprache her, die ich kurzerhand erfand [...]"
- „mit Anordung des/r Widersta/änd/-e"
- „Ich bin dabei nicht systematisch vorgegangen."
- „Das plazieren [sic] der Widerstände im Steckbret [sic] und das errechnen des Gesammtwiderstandes [sic] eines Steckbrets."
- „Quality of Life Funktionen"