Lösungshinweise und Bewertungskriterien

Allgemeines

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

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

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

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

Einige Anmerkungen noch zur Bewertung:

- Pro Aufgabe werden maximal fünf Punkte vergeben. In der Hauptliga sind für die Gesamtbewertung die drei am besten bewerteten Aufgabenlösungen maßgeblich; es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreicht man mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten und einen 3. Preis mit 9 bis 11 Punkten. Träger eines 1. oder 2. Preises sind für die zweite Runde qualifiziert.
- In der Juniorliga wird ein 1. Preis für 9 oder 10 Punkte, ein 2. Preis für 7 oder 8 Punkte und ein 3. Preis für 5 oder 6 Punkte vergeben. Leider gibt es in der Juniorliga (noch) keine zweite Runde.
- Eine Einsendung wird in Juniorliga und Hauptliga gewertet, wenn alle Gruppenmitglieder die Altersbedingung für Junioraufgaben erfüllen (damit kommt die Einsendung für die Juniorliga in Frage) und in der Einsendung sowohl Junioraufgaben als auch andere Aufgaben bearbeitet wurden.
- Auf den Bewertungsbögen bedeutet ein Kreuz in einer Zeile, dass die (meist negative) Aussage in dieser Zeile auf die Einsendung zutrifft. Damit verbunden ist dann in der Regel der Abzug eines oder mehrerer Punkte. Eine Wellenlinie ($\sim$) bedeutet „na ja, hätte besser sein können“, führt aber alleine nicht zu Punktabzug. Mehrere Wellenlinien können sich aber zu einem Punktabzug addieren. Gelegentlich sind lobende Anmerkungen der Bewerter mit einem '+' versehen.
- Leider ließ sich nicht verhindern, dass etliche Teilnehmer nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden. Das ist ziemlich riskant, da Fehler sich leicht einschleichen.
- Grundsätzlich kann die Dokumentation als „schlecht / nicht vollständig“ bewertet werden, wenn Teile wie Umsetzung, Beispiele oder Quellcode(Auszüge) fehlen. Außerdem wird in der Regel gefordert, dass die gewählten Verfahren gut nachvollziehbar beschrieben sind und begründet wird, warum sie das gegebene Problem lösen.

Zum Schluss:

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

Danksagung

An der Erstellung der Lösungsideen haben mitgewirkt: Felix Frei (Junioraufgabe 1), Philip Rehorst, Daniel Schmidt und Melanie Schmidt (Junioraufgabe 2), Robert Czechowski und Nikolai Wyderka (Aufgabe 1), Thekla Hamm (Aufgabe 2), Meike Grewing (Aufgabe 3), Martin Thoma (Aufgabe 4) und Philip Wellnitz (Aufgabe 5).

Die Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt, und zwar aus Vorschlägen von Wolfgang Pohl (Junioraufgabe 1), Philip Rehorst, Daniel Schmidt und Melanie Schmidt (Junioraufgabe 2), Peter Rossmanith (Aufgaben 1 und 3), Torben Hagerup (Aufgabe 2), Holger Schlingloff (Aufgabe 4) und Jens Gallenbacher (Aufgabe 5).

J1 Fähre füllen

Eine erfolgreiche Lösung dieser Aufgabe enthält 3 Teile:

(1) Einlesen der Daten.

(2) Entwicklung und Implementierung der Strategien.

(3) Übersichtliche Ausgabe der Ergebnisse.

Zunächst werden die Teile (1) und (3) kurz behandelt. Der wichtige und interessante Teil der Aufgabe ist aber Punkt (2), auf den anschließend ausführlich eingegangen wird.

J1.1 Einlesen der Daten

Das Programm muss in der Lage sein, eine Folge von Zahlen einzulesen, die alle in einer Datei stehen und jeweils durch einzelne Leerzeichen getrennt sind. Die Eingabe hat also das folgende Format:

$$Fahrzeuglaenge_1 \sqcup Fahrzeuglaenge_2 \sqcup \dots \sqcup Fahrzeuglaenge_n$$

wobei jede einzelne Zahl maximal 2 Nachkommastellen (getrennt von den 'Vorkommastellen' durch Punkte) hat. Da jede Parkbahn genau 20m lang ist, können wir davon ausgehen, dass jedes Fahrzeug nicht länger als 20m ist. Also reicht eine normale Fließkommazahl, um eine Fahrzeuglänge abzuspeichern. Alternativ kann die Länge in Zentimetern als Integer gespeichert werden. Da nicht von vornherein bekannt ist, wie viele Fahrzeuge zu parken sind, müssen wir eine Datenstruktur verwenden, deren Größe zum Start des Programms noch nicht feststehen muss. Hier bieten sich Listen oder dynamische Arrays an.

Alternativ hilft folgende Beobachtung: eine Datei kann zwar beliebig viele Fahrzeuge enthalten, aber da zwischen 2 Fahrzeugen je 30cm Platz gelassen werden muss, können nur begrenzt viele Fahrzeuge auf der Fähre Platz finden. Da $3 \cdot 20/0.3 = 200$, ist ein Array mit 200 Zahlen auf jeden Fall ausreichend, um genügend Fahrzeuglängen abzuspeichern[1].

J1.2 Übersichtliche Ausgabe der Ergebnisse

Besonders schön wäre eine grafische Benutzeroberfläche, auf der man direkt erkennt, wie lang die einzelnen Fahrzeuge sind. Diese hätte den Vorteil, dass intuitiv ersichtlich ist, welche Fahrzeuge wie viel Platz brauchen (auch im Vergleich untereinander), wie viel Platz auf jeder Parkbahn am Ende noch frei ist und wohin man welche Fahrzeuge noch verschieben könnte. Eine textbasierte Ausgabe in der Konsole reicht aber vollkommen. Dabei sollten folgende Kriterien erfüllt sein: Es muss deutlich werden,

- welches Fahrzeug in welcher Parkbahn steht und
- in welcher Reihenfolge die Fahrzeuge in einer Parkbahn stehen.

Gut ist außerdem, wenn der noch freie Platz (oder der schon belegte Platz) auf jeder Parkbahn angegeben wird. Die Ausgabe das Programms kann dann z.B. so aussehen:

[1] Tatsächlich reichen noch weniger. Warum?

```
P1: Rückseite | 4.75 3.12 | Auffahrt Freier Platz=11.53
P2: Rückseite | 5.89 2.77 6.16 | Auffahrt Freier Platz=4.28
P3: Rückseite | 16.44 | Auffahrt Freier Platz=3.26
```

Die Wörter 'Rückseite' und 'Auffahrt' verdeutlichen, auf welcher Seite der Parkbahn die Fahrzeuge auf die Fähre fahren. Zusätzlich können ein oder mehrere Zeichen eingesetzt werden, um den Trenn-Platz zwischen den einzelnen Fahrzeugen anzuzeigen, z.B. mit || oder |0.3|.

Wenn die Ausgabe so realisiert wird, kann aber ein Problem auftreten, wenn zwei Fahrzeuge die gleiche Länge haben. Hierzu ein Eingabe-Beispiel: `2.5 3.1 3.1 3.4 5.4`

In der Eingabe ist jedes Fahrzeug durch seine Position in der Warteschlange eindeutig identifiziert. Ein Programm könnte die ersten drei Fahrzeuge folgendermaßen aufteilen:

```
P1: 2.5
P2: 3.1
P3: 3.1
```

Jetzt ist nicht zu erkennen, ob das zweite (bzw. dritte) Fahrzeug in P2 oder P3 eingeordnet wurde. Dieser Mangel kann behoben werden, indem zusätzlich zum Endergebnis die Ausgabe der Zuordnungsstrategie angezeigt wird, z.B. so:

```
2.5->P1, 3.1->P2, 3.1->P3 ...
```

Alternativ kann die Position eines Fahrzeugs in der Warteschlange direkt in die Darstellung der Parkbahnbelegung eingefügt werden. Das Ergebnis könnte so aussehen:

```
P1: Rückseite | 2.5[1] || 8.42[5] | Auffahrt Belegt=11.22
P2: Rückseite | 3.1[2] || 1.77[4] || 6.66[6] | Auffahrt Belegt=12.13
P3: Rückseite | 3.1[3] || 1.77[7] | Auffahrt Belegt=5.17
```

J1.3 Die Strategien: Grundlagen

Mathematische Modellierung

Der erste Schritt einer Problemlösung in der Informatik ist eine Modellierung. Das bedeutet, dass man von allen Details abstrahiert, die für die Lösung nicht wichtig sind, sodass man sich auf das Kernproblem konzentrieren kann. Zum Beispiel wird in dieser Aufgabe ein Fährbegleiter genannt. Dieser spielt für die Aufgabe aber keine große Rolle, sodass wir ihm keine besondere Aufmerksamkeit schenken sollten. Außerdem gibt es auf der Fähre verschiedene Fahrzeuge: PKWs mit Anhänger, PKWs ohne Anhänger, Kleinlaster und andere. Diese Unterscheidung spielt für uns aber ebenfalls keine Rolle; uns interessiert nur die Länge der Fahrzeuge.

Insbesondere kann eine mathematische Modellierung hilfreich sein, bei der man die wesentlichen Dinge in der Sprache der Mathematik beschreibt: Die Eingabe für das Programm ist eine Folge

$$f_1\, f_2 \cdots f_n$$

und es gilt $n \in \{1, 2 \ldots 200\}$, und für alle i ist $0 < f_i \leq 20$ und f_i ist auf 2 Nachkommastellen genau (bzw. alternativ in Zentimetern als natürliche Zahl) angegeben. Die Ausgabe des Programms ist eine Zuordnung der Fahrzeuge zu ihren Parkbahnen, formal also eine Folge

$$p_1\, p_2 \cdots p_m$$

mit $p_i \in \{1,2,3\}$ für alle i. Da in vielen Fällen nicht alle Fahrzeuge auf die Fähre passen, gilt $m \leq n$.

Weitere Beobachtungen: Jede Parkbahn ist 20m lang, und der Mindestabstand zwischen zwei Fahrzeugen beträgt 30cm. Für eine Ausgabe $p_1, \dots p_m$ muss also gelten, für jede Parkbahn $b \in \{1,2,3\}$

$$\sum_{\substack{i \in \{1 \dots m\} \\ p_i = b}} f_i + 0.30 \leq 20 + 0.30$$

Außerdem können maximal 66 Fahrzeuge auf jede Parkbahn zugewiesen werden[2], denn $\frac{20}{0.3} = 66\frac{2}{3} < 67$, also gilt formal für alle $b \in \{1,2,3\}$

$$|\{i | p_i = b\}| \leq 66$$

Vergleich informatischer Probleme

Viele Aufgaben beim Bundeswettbewerb Informatik beruhen auf Informatik-Problemen aus der *echten Welt*™. Das vorliegende Problem kann als Variante des Load-Balancing-Problems betrachtet werden. Dabei bekommt ein Betriebssystem nacheinander Prozesse zugewiesen und muss für jeden Prozess einen Prozessor aussuchen, der den Prozess abarbeitet. Das Problem ist auch unter dem Namen 'Job Shop Scheduling' bekannt[3].

Online-Algorithmen

Nehmen wir mal an, der Fährbegleiter wäre etwas größer, und könnte nicht nur das aktuelle, sondern auch alle folgenden Autos sehen und deren Länge kennen. Dann wäre es einfach, das Problem zu lösen. Man müsste nur alle Möglichkeiten ausprobieren[4], wie man die Fahrzeuge einsortieren könnte, und würde schließlich die beste finden.

Interessant wird das Problem aber dadurch, dass der Fährmann immer nur die Länge des aktuellen Autos kennt und nichts über die restlichen Autos weiß, nicht mal, wie viele noch kommen werden. Es handelt sich bei dieser Aufgabe also um ein Online-Problem[5], da die Eingabedaten während der Ausführung des Algorithmus nur schrittweise bekannt gegeben werden. Daher können wir auch nicht erwarten, für dieses Problem einen Algorithmus zu finden, der immer die beste Lösung berechnet. Das folgende, einfache Beispiel beweist schon, dass das nicht funktionieren kann:

Betrachte die folgenden 2 Eingaben:

```
(1) 20 9 9  2 10 10
(2) 20 9 9 11 11
```

[2] Das ist keine strenge obere Schranke; Die tatsächliche maximale Anzahl liegt sogar noch darunter, weil jedes Fahrzeug mindestens die Länge 0.01 hat.

[3] `http://en.wikipedia.org/wiki/Job_shop_scheduling`

[4] Don't try this at home ;-) `http://de.wikipedia.org/wiki/Brute-Force-Methode`

[5] `http://de.wikipedia.org/wiki/Online-Algorithmus`

Nachdem der Fährmann die ersten beiden Autos zugeordnet hat, hat er für das dritte Auto genau zwei Möglichkeiten. Wenn er es zu dem ersten Auto mit Länge 9 zuordnet und Eingabe (1) folgt, war die Entscheidung richtig. Wenn aber Eingabe (2) folgt, war die Entscheidung falsch. Alle Möglichkeiten werden in der folgenden Tabelle dargestellt:

<table>
<tr><th>Zuordnung der Fahrzeuge 1-3</th><th>Endergebnis bei Eingabe (1)</th><th>Endergebnis bei Eingabe (2)</th></tr>
<tr><td>20→ P1, 9 → P2, 9→P2</td><td><code>P1: | 20 |</code>
<code>P2: | 9 || 9 || 2|</code>
<code>P3: | 10 || 10|</code></td><td><code>P1: | 20 |</code>
<code>P2: | 9 || 9 |</code>
<code>P3: | 11 |</code></td></tr>
<tr><td>20→ P1, 9 → P2, 9→P3</td><td><code>P1: | 20 |</code>
<code>P2: | 9 || 2|</code>
<code>P3: | 9 || 10|</code></td><td><code>P1: | 20 |</code>
<code>P2: | 9 || 11 |</code>
<code>P3: | 9 || 11 |</code></td></tr>
</table>

Bei einem Online-Algorithmus wird oft untersucht, wie gut dessen Lösungen sind im Vergleich zu den Lösungen eines optimalen „Offline-Algorithmus“, der die Eingabedaten komplett kennt. Das Verhältnis zwischen Online- und Offline-Lösungen heißt „kompetitiver Faktor“, ist im Rahmen dieser Aufgabe aber nicht weiter wichtig.

J1.4 Die Strategien: Beispiele

Diese Aufgabe bietet Raum für Kreativität. Man kann sich viele verschiedene Strategien ausdenken, die je nach Eingabe unterschiedlich gute Ergebnisse bringen. Es wurde nicht erwartet, eine besonders gute Strategie zu finden, denn zu jeder Strategie lässt sich eine Eingabe konstruieren, für die diese Strategie nicht so gut abschneidet. Im Folgenden sind ein paar Ideen für verschiedene Strategien vorgestellt, außerdem Beispiele, für die diese Strategien gut oder schlecht abschneiden. Um die Tabellen übersichtlich zu halten, wird nicht überall die Position der Fahrzeuge in der Warteschlange im Endergebnis angegeben.

Strategie „Der Reihe nach“

Die Idee dieser Strategie ist, die Parkbahnen nacheinander aufzufüllen. Erst wenn eine Parkbahn komplett voll ist, wird das kommende Fahrzeug auf die nächste Parkbahn geschickt. Algorithmus 1 beschreibt die Strategie in Pseudocode:

Algorithmus 1 'Der Reihe Nach'

```
for all Fahrzeuge F aus der Warteschlange do
    if F passt in Parkbahn1 then
        Parkbahn1 ← F
    else if F passt in Parkbahn2 then
        Parkbahn2 ← F
    else if F passt in Parkbahn3 then
        Parkbahn3 ← F
    else
        Fahrzeug F passt nicht mehr.
    end if
end for
```

Die folgende Tabelle zeigt, dass diese Strategie manche Eingaben optimal zuteilt, aber bereits das Ändern der Fahrzeugreihenfolge die optimale Zuteilung verhindern kann:

Eingabe	`9 10.7 9 10.7 9 10.7`	`9 9 9 10.7 10.7 10.7`

Ergebnis	`P1: | 9[1] 10.7[2] |`
	`P2: | 9[3] 10.7[4] |`
	`P3: | 9[5] 10.7[6] |`
		`P1: | 9[1] 9[2] |`
		`P2: | 9[3] 10.7[4] |`
		`P3: | 10.7[5] |`

Bewertung	Optimale Zuteilung.	Letztes Fahrzeug passt nicht mehr drauf :(

Strategie „Balanciert“

Diese Strategie versucht, den freien Platz in allen Parkbahnen immer möglichst gleich zu halten. Dafür wird jedes Fahrzeug immer in die Parkbahn mit dem größten freien Platz geschickt. Im folgenden liefert die Funktion „maxFrei(P)“ für eine Menge P von Parkbahnen den Index der am wenigsten belegten Parkbahn.

Algorithmus 2 'Balanciert'

```
for all Fahrzeuge F aus der Warteschlange do
    X ← maxFrei([1,2,3])
    if F passt in ParkbahnX then
        ParkbahnX ← F
    else
        Fahrzeug F passt nicht mehr.
    end if
end for
```

Eingabe	`9 9 9 10.7 10.7 10.7`	`5 5 5 16 2 3 5 4 4 6`

Ergebnis	`P1: | 9[1] 10.7[4] |`
	`P2: | 9[2] 10.7[5] |`
	`P3: | 9[3] 10.7[6] |`
		`P1: | 5[1] |`
		`P2: | 5[2] |`
		`P3: | 5[3] |`

Bewertung	Gutes Ergebnis: Optimale Zuteilung.	Schlecht: Das lange Fahrzeug(16) passt bereits nicht mehr. Bei intelligenter Zuteilung hätten alle Fahrzeuge auf die Fähre gepasst.

Strategie „Balanciert, kurze Kasse“

Die Idee dieser Strategie ist vielleicht von Supermärkten bekannt, wo nur Kunden mit bis zu 10 Artikeln sich an einer sogenannten 'Schnellkasse' anstellen dürfen. Analog wird hier die dritte Parkbahn für kurze Fahrzeuge reserviert. Die anderen beiden Parkbahnen werden wie bei der Strategie „Balanciert“ behandelt.

Algorithmus 3 'Balanciert, kurze Kasse'

```
for all Fahrzeuge F aus der Warteschlange do
    if F < 4.00 then
        if F passt in Parkbahn3 then
            Parkbahn3 ← F
        else
            Versuche, F beliebig in Parkbahn 1 oder 2 einzuordnen.
        end if
    else
        X ← maxFrei([1,2])
        if F passt in ParkbahnX then
            ParkbahnX ← F
        else
            Versuche, F in Parkbahn 3 einzuordnen.
        end if
    end if
end for
```

Eingabe	`3 3 3 3 3 3 18 17 2`	`16 16 3 3 3 12 2 2 2 2`

Ergebnis	`P1:| 18[7] |`
	`P2:| 17[8] 2[9] |`
	`P3:|3[1] 3 3 3 3 3[6]|`
		`P1: | 16[1] |`
		`P2: | 16[2] |`
		`P3: |3[3] 3 3|`

Bewertung	Gutes Ergebnis: Optimale Zuteilung.	Schlecht: Das Fahrzeug mit Länge 12 passt nicht mehr drauf. Die Strategie „Der Reihe nach“ hätte dagegen alle Fahrzeuge auf die Fähre bekommen.

Strategie „2 Balanciert, 1 Reserviert“

Bei der Untersuchung der letzten beiden Strategien wird deutlich, dass es manchmal ein Problem ist, wenn keine Bahn komplett leer bleibt, da dann das nächste Fahrzeug mit großer Länge nicht mehr drauf passt. Ausgehend von dieser Beobachtung lässt sich folgende Strategie entwickeln:

Algorithmus 4 '2 Balanciert, 1 Reserviert'

```
for all Fahrzeuge F aus der Warteschlange do
    X = maxFrei([1,2])
    if F passt in ParkbahnX then
        ParkbahnX ← F
    else if F passt in Parkbahn 3 then
        Parkbahn3 ← F
    else
        Fahrzeug F passt nicht mehr.
    end if
end for
```

Das Ziel dieser Strategie ist also, Parkbahn 3 möglichst lange frei zu halten. Aber manchmal ist auch das insgesamt keine gute Idee:

Eingabe	`4 4 8 8 17 2 5`	`3 3 3 16 16 16`

Ergebnis	`P1:| 4[1] 8[3] 2[6] |`
	`P2:| 4[2] 8[4] 5[7] |`
	`P3:| 17[5] |`
		`P1:| 3[1] 3[3] |`
		`P2:| 3[2] 16[4] |`
		`P3:| 16[5] |`

Bewertung	Gutes Ergebnis: Optimale Zuteilung.	Schlecht: Das letzte Fahrzeug passt nicht mehr. Die einfache Strategie „Balanciert“ hätte dagegen alle Fahrzeuge verladen.

Strategie „Klassifizierung“

Bei der Strategie „Balanciert, kurze Kasse“ werden die Fahrzeuge je nach Länge unterschiedlich behandelt. Diese Idee wird mit der Strategie „Klassifizierung“ noch weiter ausgearbeitet:

Algorithmus 5 'Klassifizierung'

Verwendet: (*) = F passt dort nicht? Dann versuche, F irgendwo anders einzusortieren.

```
for all Fahrzeuge F aus der Warteschlange do
    if F > 8 then
        Parkbahn3 ← F (*)
    else if 5 < F ≤ 8 then
        Parkbahn2 ← F (*)
    else if F ≤ 5 then
        Parkbahn1 ← F (*)
    end if
end for
```

Die Idee der Strategie ist also, Fahrzeuge in die drei Klassen $K_1 = \{F|\text{Länge}(F) \leq 5\}$, $K_2 = \{F|5 < \text{Länge}(F) \leq 8\}$ und $K_3 = \{F|\text{Länge}(F) > 8\}$ einzuteilen, wobei $K = K_1 \cup K_2 \cup K_3$ die Menge aller Fahrzeuge ist. Es scheint keinen überzeugenden Grund zu geben, warum die Strategie gut funktionieren sollte; aber es ist wiederum nicht schwer, Beispiele zu konstruieren, für die sie gut bzw. schlecht funktioniert:

Eingabe	`4 6 8 4 6 8 4 4 6 3`	`4 6 8 8 18 2 3 5`

Ergebnis	`P1|4[1] 4[4] 4[7] 4[8]|`
	`P2| 6[2] 6[5] 6[9] |`
	`P3| 8[3] 8[6] 3[10] |`
		`P1:| 4[1] |`
		`P2:| 6[2] |`
		`P3:| 8[3] 8[4] |`

Bewertung	Gutes Ergebnis: Optimale Zuteilung.	Schlecht: Das große Fahrzeug (18) passt nicht. Die Strategie „Der Reihe Nach" hätte alle Fahrzeuge verladen.

Strategie „Random"

Nachdem man eine Weile versucht hat, gute Strategien zu entwickeln, stellt man fest, dass sie alle in bestimmten Situationen nicht gut funktionieren. Dummerweise weiß man nicht, welche Situationen in der Realität oft auftauchen. Es könnte ja sein, dass man eine Strategie entwickelt, die nur in einer Situation nicht gut funktioniert, aber dann kommt immer genau diese Situation in der Realität vor. Das wäre schlecht. Um diesen Fall zu vermeiden, kann man die Strategie „Random" einsetzen:

Algorithmus 6 'Random'

```
Verwendet: Rand() liefert eine äußerst zufällige Zahl aus einer Menge
  for all Fahrzeuge F aus der Warteschlange do
      if F passt in keine Parkbahn mehr then
          break
      else
          loop
              X = Rand([1,2,3])
              if F passt in ParkbahnX then
                  ParkbahnX ← F
                  break
              end if
          end loop
      end if
  end for
```

Durch wiederholte Anwendung kann man diese Strategie auch benutzen, um gute Lösungen zu finden, wenn man keine Lust hat, nachzudenken.

J1.5 Weiterführende Ideen zu Strategien

Die vorgestellten sechs Strategien sind lediglich Anregungen. Viele andere sind denkbar. Hier werden noch ein paar weitere Ideen vorgestellt, wie man neue Strategien entwickeln oder bisherige modifizieren könnte.

Best-Fit Im letzten Kapitel kam vielleicht der Eindruck auf, dass jede Zuordnung eines Fahrzeugs in eine Parkbahn die falsche Entscheidung sein könnte, je nachdem, welche Fahrzeuge danach noch folgen. Tatsächlich gibt es aber einen Fall, für den immer die optimale Zuordnung bestimmt werden kann: Wenn ein Fahrzeug F mit Länge $|F|$ ansteht, und es eine Parkbahn P gibt, die zu $|P|$ gefüllt ist, und $|P| + |F| + 0,30 = 20$, dann ist es immer optimal, F in P zu parken. Mit dieser kleinen Änderung können alle vorgestellten Strategien noch verbessert werden. Möglicherweise erhält man eine brauchbare Heuristik, wenn man die Regel so abwandelt, dass man ein Fahrzeug F der Parkbahn P zuweist, wenn $0 < 20 - |P| + |F| < \delta$, und für δ einen brauchbaren Wert ermittelt.

Hellsehen und Beobachtung Durch das Sammeln von Daten in der echten Welt könnte eine Strategie entwickelt werden, die auf Basis eines stochastischen Modells entscheidet, mit welcher Wahrscheinlichkeit welche Fahrzeuge noch kommen. Die Beispiel-Eingabedaten legen z. B. die Vermutung nahe, dass Fahrzeuge grundsätzlich über 2,5m lang sind. Die Beobachtung von noch mehr Fahrzeugen könnte diese Vermutung erhärten. Das Wissen über diese Tatsache könnte bestehende Strategien weiter verbessern. Somit wüsste dann eine Strategie, dass jede Parkbahn, die zu 17,2m befüllt ist, keine weiteren Fahrzeuge mehr aufnehmen kann.

Kombination von Parkbahn-Füllstand und Fahrzeuglänge Die oben vorgestellten Strategien entscheiden die Wahl der Parkbahn entweder abhängig von der Fahrzeuglänge oder abhängig vom Füllstand der einzelnen Parkbahnen. Diese Einschränkung auf ein Kriterium ist natürlich nicht notwendig: man könnte Strategien entwickeln, die beide Faktoren mit einbeziehen. Zum Beispiel so:

Algorithmus 7 'Kombinationsalgorithmusentwurf'

```
for all Fahrzeuge F aus der Warteschlange do
    if Alle Parkbahnen sind maximal 10m weit gefüllt then
        Strategie „Balanciert“ anwenden
    else
        Strategie „Der Reihe Nach“ anwenden.
    end if
end for
```

J1.6 Ergebnisse

Es wurde ein Programm (in der Sprache Ruby) entwickelt, das alle vorgestellten Strategien implementiert. In diesem Abschnitt wird die Ausgabe des Programms bei Eingabe der Beispieldaten abgedruckt. Der Aufruf des Programms erfolgt in der Konsole mit dem Befehl:

```
ruby faehrefuellen.rb "faehre-beispielX.txt"
```

Um die Ausgabe kompakter abzudrucken, wurde sie hier etwas gekürzt; 'Px' steht für 'Parkbahn mit Nummer x', 'R' steht für 'Rückseite', 'A' steht für 'Auffahrt' und 'B' steht für 'Belegter Platz'

Ausgabe für „faehre-beispiel1.txt“

```
33. BWInf, Junioraufgabe 1: Faehre fuellen. Herzlich willkommen bei
   der Musterloesung.
13 Fahrzeuge eingelesen: 6.96 5.06 3.77 3.95 3.91 3.54 4.26 4.03 5.43
    4.04 4.43 4.12 2.78

Ergebnis von Strategie 'Balanciert':
P1: R| 6.96[1] 3.54[6] 5.43[9] |A B=16.53
P2: R| 5.06[2] 3.91[5] 4.03[8] 4.43[11] |A B=18.33
P3: R| 3.77[3] 3.95[4] 4.26[7] 4.04[10] |A B=16.92
Anzahl verstauter Fahrzeuge: 11 von 13

Ergebnis von Strategie 'Balanciert, kurze Kasse, mit Threshold 4.0':
P1: R| 6.96[1] 4.03[8] 4.04[10] |A B=15.63
P2: R| 5.06[2] 4.26[7] 5.43[9] |A B=15.35
P3: R| 3.77[3] 3.95[4] 3.91[5] 3.54[6] |A B=16.07
Anzahl verstauter Fahrzeuge: 10 von 13

Ergebnis von Strategie '2 Balanciert, 1 Reserviert':
P1: R| 6.96[1] 3.95[4] 3.54[6] 4.03[8] |A B=19.38
P2: R| 5.06[2] 3.77[3] 3.91[5] 4.26[7] |A B=17.9
P3: R| 5.43[9] 4.04[10] 4.43[11] 4.12[12] |A B=18.92
Anzahl verstauter Fahrzeuge: 12 von 13

Ergebnis von Strategie 'Der Reihe Nach':
P1: R| 6.96[1] 5.06[2] 3.77[3] |A B=16.39
P2: R| 3.95[4] 3.91[5] 3.54[6] 4.26[7] |A B=16.56
P3: R| 4.03[8] 5.43[9] 4.04[10] 4.43[11] |A B=18.83
Anzahl verstauter Fahrzeuge: 11 von 13

Ergebnis von Strategie 'Klassifizierung':
P1: R| 4.26[7] 4.03[8] 4.04[10] 4.43[11] |A B=17.66
P2: R| 6.96[1] 5.06[2] 5.43[9] |A B=18.05
P3: R| 3.77[3] 3.95[4] 3.91[5] 3.54[6] |A B=16.07
Anzahl verstauter Fahrzeuge: 11 von 13

Ergebnis von Strategie 'random':
P1: R| 6.96[1] 3.91[5] 4.03[8] 4.04[10] |A B=19.84
P2: R| 5.06[2] 5.43[9] 4.43[11] 4.12[12] |A B=19.94
P3: R| 3.77[3] 3.95[4] 3.54[6] 4.26[7] 2.78[13] |A B=19.5
Anzahl verstauter Fahrzeuge: 13 von 13

Danke fuer Ihren Besuch. Ciao.
```

Ausgabe für „faehre-beispiel2.txt“

```
33. BWInf, Junioraufgabe 1: Faehre fuellen. Herzlich willkommen bei
   der Musterloesung.
8 Fahrzeuge eingelesen: 4.14 3.63 3.92 7.95 5.23 3.3 4.86 15.06

Ergebnis von Strategie 'Balanciert':
P1: R| 4.14[1] 3.3[6] 4.86[7] |A B=12.9
P2: R| 3.63[2] 7.95[4] |A B=11.88
P3: R| 3.92[3] 5.23[5] |A B=9.45
Anzahl verstauter Fahrzeuge: 7 von 8

Ergebnis von Strategie 'Balanciert, kurze Kasse, mit Threshold 4.0':
P1: R| 4.14[1] 5.23[5] |A B=9.67
P2: R| 7.95[4] 4.86[7] |A B=13.11
P3: R| 3.63[2] 3.92[3] 3.3[6] |A B=11.45
Anzahl verstauter Fahrzeuge: 7 von 8

Ergebnis von Strategie '2 Balanciert, 1 Reserviert':
P1: R| 4.14[1] 7.95[4] 3.3[6] |A B=15.99
P2: R| 3.63[2] 3.92[3] 5.23[5] 4.86[7] |A B=18.54
P3: R| 15.06[8] |A B=15.06
Anzahl verstauter Fahrzeuge: 8 von 8

Ergebnis von Strategie 'Der Reihe Nach':
P1: R| 4.14[1] 3.63[2] 3.92[3] 5.23[5] |A B=17.82
P2: R| 7.95[4] 3.3[6] 4.86[7] |A B=16.71
P3: R| 15.06[8] |A B=15.06
Anzahl verstauter Fahrzeuge: 8 von 8

Ergebnis von Strategie 'Klassifizierung':
P1: R| 4.86[7] |A B=4.86
P2: R| 7.95[4] 5.23[5] |A B=13.48
P3: R| 4.14[1] 3.63[2] 3.92[3] 3.3[6] |A B=15.89
Anzahl verstauter Fahrzeuge: 7 von 8

Ergebnis von Strategie 'random':
P1: R| 4.14[1] 3.63[2] |A B=8.07
P2: R| 7.95[4] 5.23[5] 4.86[7] |A B=18.64
P3: R| 3.92[3] 3.3[6] |A B=7.52
Anzahl verstauter Fahrzeuge: 7 von 8

Danke fuer Ihren Besuch. Ciao.
```

Ausgabe für „faehre-beispiel3.txt“

```
33. BWInf, Junioraufgabe 1: Faehre fuellen. Herzlich willkommen bei
   der Musterloesung.
11 Fahrzeuge eingelesen: 5.23 4.41 3.33 13.13 9.12 4.38 6.34 5.37
   4.11 3.74 10.62
```

```
Ergebnis von Strategie 'Balanciert':
P1: R| 5.23[1] 4.38[6] 6.34[7] |A B=16.55
P2: R| 4.41[2] 9.12[5] 5.37[8] |A B=19.5
P3: R| 3.33[3] 13.13[4] |A B=16.76
Anzahl verstauter Fahrzeuge: 8 von 11

Ergebnis von Strategie 'Balanciert, kurze Kasse, mit Threshold 4.0':
P1: R| 5.23[1] 9.12[5] 4.38[6] |A B=19.33
P2: R| 4.41[2] 13.13[4] |A B=17.84
P3: R| 3.33[3] 6.34[7] 5.37[8] |A B=15.64
Anzahl verstauter Fahrzeuge: 8 von 11

Ergebnis von Strategie '2 Balanciert, 1 Reserviert':
P1: R| 5.23[1] 13.13[4] |A B=18.66
P2: R| 4.41[2] 3.33[3] 9.12[5] |A B=17.46
P3: R| 4.38[6] 6.34[7] 5.37[8] |A B=16.69
Anzahl verstauter Fahrzeuge: 8 von 11

Ergebnis von Strategie 'Der Reihe Nach':
P1: R| 5.23[1] 4.41[2] 3.33[3] 4.38[6] |A B=18.25
P2: R| 13.13[4] 6.34[7] |A B=19.77
P3: R| 9.12[5] 5.37[8] 4.11[9] |A B=19.2
Anzahl verstauter Fahrzeuge: 9 von 11

Ergebnis von Strategie 'Klassifizierung':
P1: R| 13.13[4] 6.34[7] |A B=19.77
P2: R| 5.23[1] 9.12[5] 4.11[9] |A B=19.06
P3: R| 4.41[2] 3.33[3] 4.38[6] 5.37[8] |A B=18.39
Anzahl verstauter Fahrzeuge: 9 von 11

Ergebnis von Strategie 'random':
P1: R| 4.41[2] 3.33[3] 9.12[5] |A B=17.46
P2: R| 13.13[4] 5.37[8] |A B=18.8
P3: R| 5.23[1] 4.38[6] 6.34[7] |A B=16.55
Anzahl verstauter Fahrzeuge: 8 von 11

Danke fuer Ihren Besuch. Ciao.
```

J1.7 Bewertungskriterien

Insgesamt muss eine Einsendung zu dieser Junioraufgabe bei weitem nicht so umfangreich sein wie diese Beispiellösung. Aber auch einfache Lösungen sollten die folgenden wesentlichen Kriterien erfüllen.

- Es müssen mindestens zwei Strategien A und B nachvollziehbar beschrieben werden, und diese müssen auf den Beispieldaten oder auf eigens angegebenen Daten unterschiedliche Ergebnisse berechnen. Manchmal muss A, manchmal B besser abschneiden. Mehr als zwei Strategien oder detaillierte Erklärungen, welchen Sinn die Strategien haben, sind schön, aber nicht erforderlich für eine Bewertung mit voller Punktzahl.

- Die Strategien müssen korrekt implementiert werden, d.h. entsprechend ihrer Beschreibung. (Kleine Implementierungsfehler in Randfällen sind aber kein KO-Kriterium). Die 30cm Abstand müssen korrekt eingehalten werden. Es muss beachtet werden, dass die 30cm nur zwischen den Fahrzeugen, nicht am Anfang oder am Ende eingehalten werden müssen.
- Die Ausgabe muss hinreichend übersichtlich sein. Es muss klar sein, welche Fahrzeuge in welcher Reihenfolge auf welchen Parkbahnen stehen. Optional ist eine Anzeige, wie viel Platz auf den Parkbahnen noch frei ist. Fahrzeuge mit gleichen Längen müssen in der Ausgabe nicht mehr unterscheidbar sein.
- Es müssen Ergebnisse für alle drei Beispieldaten dokumentiert werden. Noch mehr Ergebnisse für noch mehr Beispieldaten sind schön, aber nicht erforderlich. Wenn stattdessen nur Ergebnisse für eigene Beispiele dokumentiert wurden, ist das auch OK (die Aufgabenstellung ist hier nicht ganz klar). In dem Fall sollten es aber mindestens drei eigene Beispiele sein. Insbesondere können die Unterschiede zwischen den Strategien auch an eigenen Beispielen gezeigt werden.

J2 Zahlenspiel

J2.1 Lösungsidee

Aufgabe erstellen

Die Aufgabe des Programms ist es, Aufgaben zur Kürzung von Brüchen zu erstellen, die, abhängig von einer gewünschten Schwierigkeit, einige gegebene Bedingungen erfüllen. Zu einer Aufgabe gehört der zu kürzende Bruch, den wir „Ausgangsbruch“ nennen wollen, und das eindeutige Ergebnis, der „Lösungsbruch“. Ein nahe liegender Lösungsansatz ist es, den Lösungsbruch zufällig zu erzeugen und diesen mit einer Reihe von Faktoren zu multiplizieren, um einen passenden Ausgangsbruch zu ermitteln. Um Aufgaben zu erhalten, welche ohne Taschenrechner zu lösen sind, sollte das Produkt dieser Faktoren selbstverständlich nicht zu hoch sein. Andernfalls würden eher Konzentration und weniger Bruchrechnung geübt.

Lösungsbruch erstellen Für die Erstellung des Bruchs werden zufällig ein Zähler sowie ein Nenner aus dem zur gegebenen Schwierigkeit passenden Zahlenraum gezogen. Die Erzeugung trivialer Brüche, bei denen Zähler und Nenner identisch sind, wird nicht zugelassen. Um den korrekten Lösungsbruch angeben zu können, muss an dieser Stelle evtl. gekürzt werden. Dafür reicht es hier, dass Zähler oder Nenner zu den Primzahlen teilerfremd sind, die Primfaktor einer Zahl im betrachteten Zahlenraum sein können. Beispiel: Im Zahlenraum $\{1, 2, \ldots, 30\}$ sind das 2, 3, 5, 7, 11 und 13. Übrigens: Man könnte auch den Ausgangsbruch erzeugen, indem man für Zähler und Nenner zufällig eine Primzahl aus dem Zahlenraum auswürfelt. Dann kennt man auf jeden Fall die korrekte Lösung. Lösungsbrüche wie $6/7$ werden dadurch aber ausgeschlossen, so dass wir uns gegen diese Variante entschieden haben.

Ausgangsbruch erstellen Der so entstandene Lösungsbruch wird nun erweitert. Dafür werden Zähler und Nenner mehrfach mit 2, 3 oder 5 multipliziert (Faktoren, die nicht prim sind, wären äquivalent zu mindestens zwei Erweiterungsschritten.). Um weder zu große noch zu kleine Brüche zu erhalten, wird mindestens zwei mal multipliziert und danach mit einer Wahrscheinlichkeit von jeweils $\frac{1}{2}$ noch so lange mit einer der drei Primzahlen erweitert bis maximal 5 mal erweitert wurde. Falls an dieser Stelle noch nicht 5 mal erweitert wurde, so wird mit einer Wahrscheinlichkeit von $\frac{1}{2}$ mit 7 erweitert. So wird der ursprüngliche Bruch maximal mit 350 erweitert. Die maximal fünf Kürzungsschritte lassen sich so noch problemlos auch von Schülern und Schülerinnen ohne Taschenrechner durchführen.

Aufgabe prüfen

Die erzeugten Aufgaben sollen von einer bestimmten Schwierigkeitsstufe sein. Die Tabelle in der Aufgabenstellung gibt für jede Schwierigkeitsstufe vier Größen vor:

1. einen Namen: leicht, mittel, schwer,
2. die „Länge“ des Ausgangsbruchs a/b,
3. einen Mindestwert für $p+q$, also für die Summe von Zähler und Nenner des Lösungsbruchs p/q, und

4. einen Höchstwert für diese Summe.

Schon während der Erzeugung der Aufgabe kann die Schwierigkeit berücksichtigt werden:

- Der Zahlenraum für Zähler p und q kann durch den Höchstwert für $p+q$ begrenzt werden.
- Ein Lösungsbruch, der die Schwierigkeitsbedingung nicht erfüllt, kann direkt verworfen werden.
- Beim Erweitern des Lösungsbruchs p/q kann laufend die Bedingung an die Länge von a/b geprüft werden.

Es genügt auch, erst nach der kompletten Erzeugung einer Aufgabe diese zu prüfen und ggf. zu verwerfen. Dann werden aber viele nicht zur Schwierigkeit passende Aufgaben unnötig erstellt.

Zur Ausgabe der gewünschten Anzahl von Aufgaben bedarf es nun nur noch einer Schleife. Außerdem sollte kontrolliert werden, dass keine Aufgabe doppelt ausgegeben wird.

J2.2 Umsetzung

Den Schwierigkeitsstufen werden Zahlen zugeordnet: $\text{leicht} = 0, \text{mittel} = 1, \text{schwer} = 2$. Die Schwierigkeitstabelle wird als geschachtelte Liste realisiert:

```
# Pro Schwierigkeit eine Liste mit: Name, Laenge a/b, Minimum p+q,
   Maximum p+q
schwierigkeitsstufen = [["leicht", 4, 0, 10],
                        ["mittel", 5, 10, 20],
                        ["schwer", 5, 20, 30]]
```

Der Zugriff auf diese Datenstruktur lässt sich durch Funktionen einkapseln, z.B.

```
def getLaenge(stufe):
    return schwierigkeitsstufen[stufe][1]
```

Außerdem werden Funktionen zur Prüfung der Schwierigkeitsbedingungen definiert, z.B.:

```
def pruefeAB(a,b,stufe):
    def laenge(n):
        return len(str(n))
    return laenge(a) + laenge(b) == getLaenge(stufe)
```

Eine einzelne Aufgabe wird durch die Funktion `erzeugeZufallsAufgabe` erzeugt. Die Aufgabe wird als Tupel `(p, q, a, b)` zurückgegeben. Als Hauptprogramm dient die Funktion `erzeugeAufgaben`. Zur Vermeidung von Dopplungen legt sie die Aufgabentupel in einer Liste ab und schlägt neue Aufgaben in dieser Liste nach.

J2.3 Beispielaufrufe

```
>>> erzeugeAufgaben(6, 0)
6 Aufgaben der Schwierigkeit leicht
Aufgabe: Kuerze  25 / 15  maximal. [Loesung:  5 / 3 ]
Aufgabe: Kuerze  42 / 70  maximal. [Loesung:  3 / 5 ]
Aufgabe: Kuerze  84 / 42  maximal. [Loesung:  2 / 1 ]
Aufgabe: Kuerze  15 / 12  maximal. [Loesung:  5 / 4 ]
Aufgabe: Kuerze  56 / 70  maximal. [Loesung:  4 / 5 ]
Aufgabe: Kuerze  10 / 25  maximal. [Loesung:  2 / 5 ]
>>> erzeugeAufgaben(6, 1)
6 Aufgaben der Schwierigkeit mittel
Aufgabe: Kuerze  100 / 70  maximal. [Loesung:  10 / 7 ]
Aufgabe: Kuerze  75 / 120  maximal. [Loesung:  5 / 8 ]
Aufgabe: Kuerze  90 / 105  maximal. [Loesung:  6 / 7 ]
Aufgabe: Kuerze  315 / 70  maximal. [Loesung:  9 / 2 ]
Aufgabe: Kuerze  135 / 60  maximal. [Loesung:  9 / 4 ]
Aufgabe: Kuerze  375 / 25  maximal. [Loesung:  15 / 1 ]
>>> erzeugeAufgaben(6, 2)
6 Aufgaben der Schwierigkeit schwer
Aufgabe: Kuerze  20 / 115  maximal. [Loesung:  4 / 23 ]
Aufgabe: Kuerze  105 / 10  maximal. [Loesung:  21 / 2 ]
Aufgabe: Kuerze  207 / 18  maximal. [Loesung:  23 / 2 ]
Aufgabe: Kuerze  210 / 98  maximal. [Loesung:  15 / 7 ]
Aufgabe: Kuerze  140 / 90  maximal. [Loesung:  14 / 9 ]
Aufgabe: Kuerze  180 / 63  maximal. [Loesung:  20 / 7 ]
>>>
```

J2.4 Bewertungskriterien

- Das Verfahren zur Aufgabenerzeugung soll nachvollziehbar beschrieben sein.
- Die ausgegebene Aufgabenliste soll folgende Bedingungen erfüllen: (a) die Liste enthält so viele Aufgaben wie gewünscht; (b) jede einzelne Aufgabe erfüllt die Bedingungen der gewünschten Schwierigkeitsstufe; (c) die Liste enthält keine Dopplungen. Nicht so schön (aber akzeptabel) ist, wenn bei vielen Aufgaben $q = 1$, der Lösungsbruch also kein echter Bruch ist.
- Jeder Ausgangsbruch a/b soll außerdem die weiteren Bedingungen der Aufgabenstellung erfüllen: $a \neq b$ und a/b ist kürzbar ($q < b$).
- Wenn man von einem Lösungsbruch ausgeht, den man zufällig ohne weitere Überlegungen zieht, z.B. Zähler und Nenner zufällig aus $\{1,\dots,20\}$, dann kann es sein, dass maximales Kürzen des Ausgangsbruchs a/b nicht diesen Lösungsbruch ergibt. Zieht man als Lösungsbruch p/q zufällig $8/20$, so lassen sich dessen Erweiterungen maximal zu $2/5$ kürzen, und nicht zu $8/20$. Einen Punktabzug gibt es dann, wenn der kürzbare Lösungsbruch auch als (vermeintliche) Lösung ausgegeben wird.
- Es sollten Beispielausgaben für alle drei Schwierigkeitsstufen dokumentiert sein.

Aufgabe 1: Faires Füllen

1.1 Lösungsidee

Auch bei dieser Aufgabe wollen wir zuerst mathematisch modellieren. Wir betrachten zwei Personen, von denen eine m und die andere n Behälter mit sich führt. Zusammen haben beide also $k = m+n$ Behälter. Die Behälter $1,\dots,m$ gehören der ersten Person, die Behälter $m+1,\dots,k$ gehören der zweiten Person. Die Größen der Behälter bezeichnen wir mit G_i ($1 \le i \le k$).

Eine *Befüllung* ist ein k-Tupel von natürlichen Zahlen $(B_1, B_2, \dots, B_k)$, wobei

$$B_i \le G_i \qquad \forall i \in \{1,\dots,k\}.$$

Eine Befüllung $(B_1, B_2, \dots, B_k)$ ist in einem Schritt in eine andere Befüllung $(B'_1, B'_2, \dots, B_k)$ *überführbar*, wenn es $i, j \in \{1,\dots,k\}$ gibt, so dass

$$B'_i - B_i = B_j - B'_j \neq 0 \qquad \text{(i wird in j umgefüllt,} \qquad (1)$$

$$B'_i = 0 \vee B'_j = G_j \qquad \text{bis i leer oder j voll ist,} \qquad (2)$$

$$B_l = B'_l \qquad \forall l \in \{1,\dots,k\} \setminus \{i,j\} \qquad \text{andere bleiben gleich.)} \qquad (3)$$

Eine *Lösung* $(L_1, L_2, \dots, L_k)$ ist eine Befüllung, in welche die Startbefüllung B^{Start} in einem oder mehreren Schritten überführbar ist und in der beide Personen eine gleich summierte Befüllung haben:

$$\sum_{i=1}^{m} L_i = \sum_{i=m+1}^{k} L_i.$$

Die letzte Bedingung können wir auch über die Gesamtbefüllung ausdrücken, die sich nach Gleichung (1) und Gleichung (3) bei keinem Umfüllungsschritt ändert, als

$$\sum_{i=1}^{m} L_i = \frac{1}{2}\sum_{i=1}^{k} L_i = \frac{1}{2}\sum_{i=1}^{k} B_i^{\text{Start}}.$$

Damit ergibt sich nun ein **Umfüllgraph**, dessen Knoten die Befüllungen sind und dessen Kanten den erlaubten Umfüllschritte entsprechen. Ein Beispiel für einen Teil eines solchen Umfüllgraphen ist in Abbildung 1 dargestellt.

Breitensuche

Um die minimale Anzahl an Umfüllschritten zu finden, wird der Umfüllgraph mit einer Breitensuche nach der Lösung durchsucht. Bei einer Breitensuche wird ein (zusammenhängender) Graph nach dem kürzesten Pfad zu einer Lösung durchsucht.

Dafür bedient man sich einer Queue-Datenstruktur (Warteschlange). Die Breitensuche erfolgt dann nach folgendem Algorithmus:

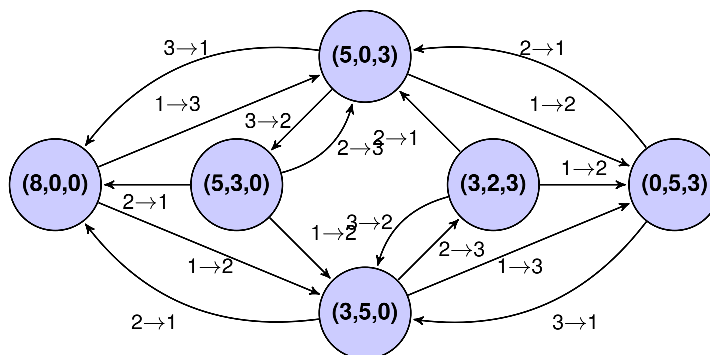


Abbildung 1: Unvollständiger Umfüllgraph für das Beispiel aus der Aufgabe und die ersten sechs erreichten Befüllungen. Die Symmetrie in diesem Graph ergibt sich durch die Erreichbarkeit des Knoten (0,5,3) für den gilt $B_i = G_i - B_i^{\text{Start}}$.

Markiere Startelement als *besucht*
Füge Startelement in die Queue ein
while Queue nicht leer **do**
    $x \leftarrow$ nächstes Element aus der Queue
    if x ist Lösung **then**
        Fertig: x ist Lösung
    end if
    for all y: von x erreichbar **do**
        if y nicht als *besucht* markiert **then**
            Füge y in die Queue ein
            Setze x als Vorgänger von y
            Markiere y als *besucht*
        end if
    end for
end while
Fertig: Keine Lösung gefunden

Der kürzeste Pfad vom Startelement zur Lösung lässt sich jetzt rekonstruieren, indem man bei der Lösung beginnt und wiederholt zum Vorgänger des aktuellen Knoten wechselt, bis das Startelement erreicht wird.

Durch Verbinden aller Knoten zu ihren Vorgängerknoten ergibt sich der Breitensuchen-Baum. Der vollständige Suchbaum für die Beispielaufgabe ist in Abb. 2 dargestellt.

1.2 Laufzeitkomplexität

Man stellt fest, dass der Algorithmus schon bei relativ kleinen Eingabegrößen sehr lange brauchen kann. In der Tat ist es möglich, dass im schlimmsten Fall bei n Behältern und einer Gesamtbefüllung von B Maßen

$$\binom{n+B-1}{n} = \frac{(n+B-1)!}{n! \cdot (B-1)!} \quad \text{Möglichkeiten}$$

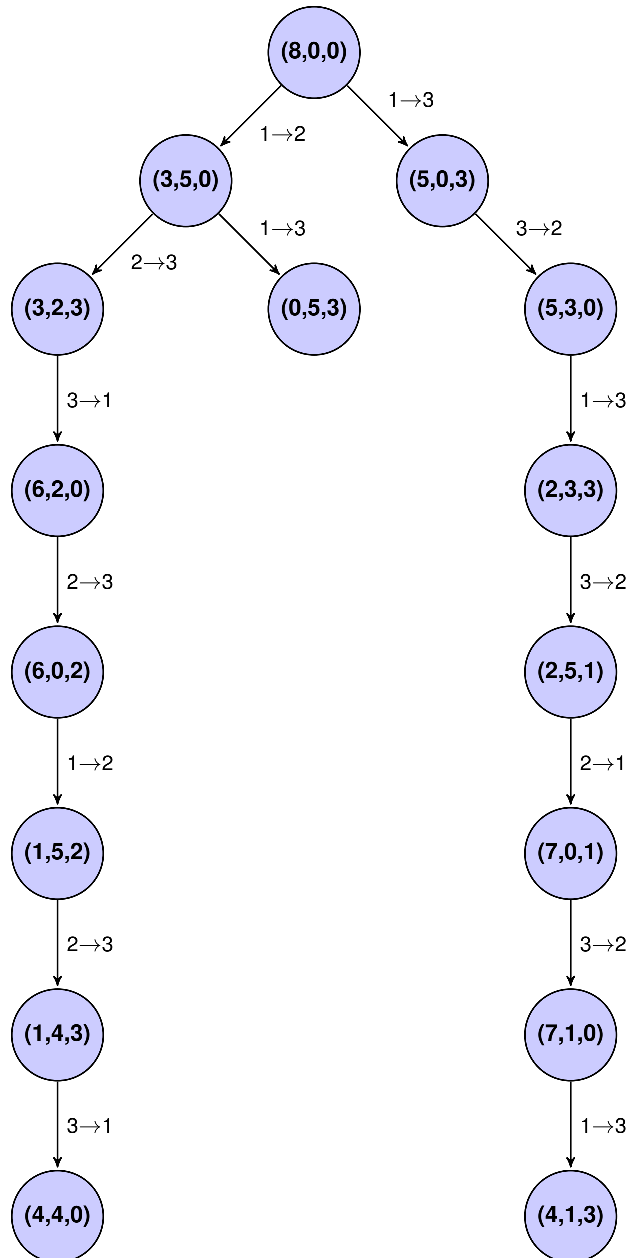


Abbildung 2: Vollständiger Breitensuchen-Baum für die Umfüllmöglichkeiten zu dem Beispiel aus der Aufgabe. Die Kantenrichtung ist hier in Richtung der Umfüllung eingezeichnet und nicht in Richtung zum Vorgänger.

durchprobiert werden müssen. Dies entspricht im Urnenmodell dem Ziehen von n Kugeln aus einer Urne mit B Kugeln mit Zurücklegen und ohne Beachten der Reihenfolge.[6] Das kann schon bei kleinen n und B eine nicht mehr bewältigbare Anzahl von Möglichkeiten ergeben.

Lässt sich die Laufzeitkomplexität verbessern? Im Allgemeinen gilt: nicht wesentlich. Das Problem ist nämlich NP-schwer, wie durch die Reduktion vom Partitionsproblem auf das Problem aus der Aufgabe gezeigt werden kann.

Partitionsproblem

Das Partitionsproblem (in der Informatik auch „PARTITION“ genannt) lautet so:

Zu einer gegebenen endlichen Menge von ganzen Zahlen $Z = \{z_1, z_2, \ldots, z_n\}$ wird eine Teilmenge $Y \subset Z$ gesucht, so dass die Summe über alle Zahlen in Y gleich der Summe über alle Zahlen aus $Z \setminus Y$ ist:

$$\sum_{y \in Y} y = \sum_{z \in Z \setminus Y} z. \tag{4}$$

Oder anders ausgedrückt, die Summe über alle Zahlen in Y soll die Hälfte der Summe über alle Zahlen in Z sein:

$$\sum_{y \in Y} y = \frac{1}{2} \sum_{z \in Z} z. \tag{5}$$

Anschaulich gesprochen soll eine Menge von Paketen mit den Gewichten z_i in zwei gleich schwere Stapel aufgeteilt werden.

Das Partitionsproblem ist bekanntermaßen NP-vollständig. Das heißt, dass es einerseits in Polynomialzeit möglich ist, zu verifizieren, ob eine Menge Y tatsächlich eine Lösung des Partitionsproblems ist, andererseits aber, dass es höchstwahrscheinlich unmöglich sein wird, einen Algorithmus zu finden, der in Polynomialzeit eine Lösung findet.[7]

Reduktion des Partitionsproblems

Gegeben sei eine Instanz des Partitionsproblem gegeben als n ganze Zahlen $z_1, \ldots, z_n$. Man definiert sich nun $(n+1)$ Behälter mit einer Größe von $\zeta := \sum_{i=1}^{n} z_i$, von denen die ersten n Behälter der ersten Person gehören und der letzte Behälter der zweiten Person. Die ersten n Behälter werden entsprechend der Zahlen aus dem Partitionsproblem gefüllt, also $B_i = z_i$, und der letzte Behälter bleibt leer.

Dadurch dass die Behälter so groß sind, dass in jeden von ihnen das ganze Flüssigkeitsvolumen ζ hineinpasst, bleiben alle Anfangsbefüllungen immer als Ganzes erhalten und können nicht aufgeteilt werden. Wenn es eine Lösung gibt, dann ist der kürzeste Pfad dorthin also, eine Teilmenge der n Behälter in den $(n+1)$-ten Behälter umzufüllen, so dass sich dort die Hälfte der Flüssigkeit befindet. Dies gibt einem dann direkt eine Lösung für das Partitionsproblem.

[6] Siehe http://de.wikipedia.org/wiki/Kombination_(Kombinatorik), Abschnitt „Kombination mit Wiederholung.“

[7] Findet man doch einen Algorithmus, so hat man das P=NP-Problem gelöst, eines der wichtigsten ungelösten Probleme der Informatik, und darf auf zahllose Ehrungen und umfangreiche Preisgelder hoffen.

Damit ist das Partitionsproblem auf das Problem auf der Aufgabe reduziert worden. Würde man jetzt eine Polynomialzeitlösung für das Problem aus der Aufgabe finden, so könnte man diese direkt nutzen, um auch das Partitionsproblem in Polynomialzeit zu lösen. Da das Partitionsproblem aber NP-vollständig ist, wäre ein Polynomialzeitagorithmus ein bahnbrechendes Ergebnis.

1.3 Umsetzung

Die Lösungsidee wurde in ein Haskell-Programm umgesetzt, das den Lösungsansatz noch ein wenig verallgemeinert. Anstelle von Befüllungen einer Reihe von Behältern wird eine beliebige Klasse von Objekten behandelt und anstelle von Umfüllungen eine beliebige Transformation dieser Objekte ineinander. Es muss ein Startobjekt angegeben werden und ein Prädikat, das entscheidet, ob ein Objekt eine Lösung des Problems ist.

Da bei in unserem Fall der Umfüllgraph nur *implizit* vorliegt, benötigen wir etwas ausgefeiltere Datenstrukturen, um uns die besuchten Knoten und die Vorgänger zu merken. Zunächst definieren wir uns eine Datenstruktur, die aus einer Map und einer Queue besteht. Die Queue dient zum Speichern der noch nicht besuchten benachbarten Knoten für die Breitensuche. Die Map dient einerseits zum schnellen Nachschlagen, ob Knoten schon besucht wurden, und andererseits dazu, jedem Objekt das Objekt zuzuordnen, aus dem es durch eine Transformation erzeugt werden kann. Konstruktionsbedingt können wir dann beim Erreichen einer Lösung einen kürzesten Weg zu dieser Lösung finden, indem wir bei der Lösung starten und wiederholt in der Vorgänger-Map nachschlagen.

Der Kern des Algorithmus ist in der Funktion `breadthFirstSearch` enthalten. Die Funktion bekommt die Transformation, das Prädikat und das Startelement übergeben.

Aus dem Startelement wird die `MapQueue`-Struktur generiert. Eine zunächst leere Liste beinhaltet die zu bearbeitenden Nachbarknoten. Ist die Liste leer, wird der nächste Knoten aus der Breitensuche-Queue der `MapQueue`-Struktur als neuer aktueller Knoten betrachtet und seine Transformation als neue Liste. Andernfalls wird die Liste rekursiv durchschritten und jedes Objekt, das noch nicht in der Vorgänger-Map vorhanden ist, wird in die `MapQueue`-Struktur eingefügt. Damit wird es einerseits in die Vorgänger-Map eingefügt (mit Verweis auf das aktuelle Objekt als Vorgänger) und anderseits in die Breitensuche-Queue für die nächsten Objekte in der Breitensuche.

Ablaufbeispiel

Das folgende Ablaufbeispiel zeigt die Inhalte der Datenstrukturen des Programms anhand des Beispiels aus der Aufgabenstellung:

Vorgänger-Map	Breitensuche-Queue	Aktuelles Element (Parent)	Transformierte Elemente
(8,0,0)→(8,0,0)	(8,0,0)		
(8,0,0)→(8,0,0)		(8,0,0)	(3,5,0) (5,0,3)
(8,0,0)→(8,0,0) (3,5,0)→(8,0,0)	(3,5,0)	(8,0,0)	(5,0,3)
(8,0,0)→(8,0,0) (3,5,0)→(8,0,0) (5,0,3)→(8,0,0)	(3,5,0) (5,0,3)	(8,0,0)	
(8,0,0)→(8,0,0) (3,5,0)→(8,0,0) (5,0,3)→(8,0,0)	(5,0,3)	(3,5,0)	(0,5,3) (8,0,0) (3,2,3)
(8,0,0)→(8,0,0) (3,5,0)→(8,0,0) (5,0,3)→(8,0,0) (0,5,3)→(3,5,0)	(5,0,3) (0,5,3)	(3,5,0)	(8,0,0) (3,2,3)
(8,0,0)→(8,0,0) (3,5,0)→(8,0,0) (5,0,3)→(8,0,0) (0,5,3)→(3,5,0)	(5,0,3) (0,5,3)	(3,5,0)	(3,2,3)
(8,0,0)→(8,0,0) (3,5,0)→(8,0,0) (5,0,3)→(8,0,0) (0,5,3)→(3,5,0) (3,2,3)→(3,5,0)	(5,0,3) (0,5,3) (3,2,3)	(3,5,0)	
(8,0,0)→(8,0,0) (3,5,0)→(8,0,0) (5,0,3)→(8,0,0) (0,5,3)→(3,5,0) (3,2,3)→(3,5,0)	(0,5,3) (3,2,3)	(5,0,3)	(0,5,3) (8,0,0) (5,3,0)
⋮	⋮	⋮	⋮

1.4 Beispiele

Für einige Beispiele sind hier nun die kürzesten Pfade zur Lösung, sofern es eine gibt, aufgeführt. Dabei wird die Ausgabe in folgendem Format dargestellt:

```
Behältergrößen
Behälterfullstände
Liste aller Füllstände auf dem Pfad vom Beginn zur Lösung
Anzahl der Schritte
```

Für die vom Bundeswettbewerb Informatik vorgegebenen Beispiele lassen sich folgende Ausgaben erzeugen:

```
 8 |  5  3
 8 |  0  0
[[8,0,0],[3,5,0],[3,2,3],[6,2,0],[6,0,2],[1,5,2],[1,4,3],[4,4,0]]
7 Schritte

10  8 | 11 7
10  4 |  4 6
[keine Lösung]

6 26 | 13 50
0  0 |  0 20
[[0,0,0,20],[6,0,0,14],[0,6,0,14],[6,6,0,8],[0,12,0,8],[6,12,0,2],
 [0,18,0,2],[0,5,13,2],[6,5,7,2],[0,11,7,2],[6,11,1,2],[0,17,1,2],
 [1,17,0,2],[1,4,13,2],[6,4,8,2]]
14 Schritte
```

Weitere Beispiele sind:

```
30 28 | 31 27
12  9 | 10 13
[[12,9,10,13],[0,9,22,13],[13,9,22,0]]
2 Schritte

30 28 | 31 27
12  9 | 11 12
[keine Lösung]

 3  5  8 13 | 21 34
 0  0  0  0 | 0 34
[[0,0,0,0,0,34],[3,0,0,0,0,31],[0,3,0,0,0,31],[3,3,0,0,0,28],
 [0,3,0,3,0,28],[3,3,0,3,0,25],[3,3,8,3,0,17]]
6 Schritte

128  64 | 32 16  8  4  2  1
  0   0 |  0  0  0  0  0  0
[[126,0,0,0,0,0,0,0],[94,0,32,0,0,0,0,0],[78,0,32,16,0,0,0,0],
 [70,0,32,16,8,0,0,0],[66,0,32,16,8,4,0,0],[64,0,32,16,8,4,2,0],
 [63,0,32,16,8,4,2,1]]
6 Schritte
```

Beispiele mit hoher Laufzeit:

```
10 10 10 10 10 10 10 | 10 10 10 10 10 10  1
10  0  0  0  0  0  0 |  0  0  0  0  0  0  0
[[10,0,0,0,0,0,0,0,0,0,0,0,0,0],[9,0,0,0,0,0,0,0,0,0,0,0,0,1],
 [9,0,0,0,0,0,0,1,0,0,0,0,0,0],[8,0,0,0,0,0,0,1,0,0,0,0,0,1],
 [8,0,0,0,0,0,0,2,0,0,0,0,0,0],[7,0,0,0,0,0,0,2,0,0,0,0,0,1],
 [7,0,0,0,0,0,0,3,0,0,0,0,0,0],[6,0,0,0,0,0,0,3,0,0,0,0,0,1],
 [6,0,0,0,0,0,0,4,0,0,0,0,0,0],[5,0,0,0,0,0,0,4,0,0,0,0,0,1]]
9 Schritte

30 30 30 | 30 30  1
30  0  0 |  0  0  0
[[30,0,0,0,0,0],[29,0,0,0,0,1],[29,0,0,1,0,0],[28,0,0,1,0,1],
 [28,0,0,2,0,0],[27,0,0,2,0,1],[27,0,0,3,0,0],[26,0,0,3,0,1],
 [26,0,0,4,0,0],[25,0,0,4,0,1],[25,0,0,5,0,0],[24,0,0,5,0,1],
 [24,0,0,6,0,0],[23,0,0,6,0,1],[23,0,0,7,0,0],[22,0,0,7,0,1],
 [22,0,0,8,0,0],[21,0,0,8,0,1],[21,0,0,9,0,0],[20,0,0,9,0,1],
 [20,0,0,10,0,0],[19,0,0,10,0,1],[19,0,0,11,0,0],[18,0,0,11,0,1],
 [18,0,0,12,0,0],[17,0,0,12,0,1],[17,0,0,13,0,0],[16,0,0,13,0,1],
 [16,0,0,14,0,0],[15,0,0,14,0,1]]
29 Schritte

 3  5  8 13 21 34 | 55 89 144
 0  0  0  0  0  0 |  0  0 144
[[0,0,0,0,0,0,0,0,144],[3,0,0,0,0,0,0,0,141],[0,3,0,0,0,0,0,0,141],
[3,3,0,0,0,0,0,0,138],[0,3,0,3,0,0,0,0,138],[3,3,0,3,0,0,0,0,135],
[3,3,8,3,0,0,0,0,127],[3,3,8,3,21,0,0,0,106],[3,3,8,3,21,34,0,0,72]]
8 Schritte
```

1.5 Sinnvolle Ergänzungen

Eine Befüllung mit einem ungeraden Maß kann nicht gerecht aufgeteilt werden. Dies kann man gleich zu Beginn erkennen. Alternativ kann man erlauben, dass eine Befüllung auch als Lösung zählt, wenn eine Person ein Maß Wein mehr bekommt. Eine solche Abwandlung muss aber explizit erwähnt werden. Außerdem sollte erwähnt werden, nach welchem Kriterium eine Person mehr bekommt als die andere (z. B. weil die Lösung zuerst gefunden wurde).

Des Weiteren kann man die Suche abkürzen, wenn die eine Person gar nicht genügend Behälter zur Verfügung hat, um die Hälfte des Weins zu transportieren. Auch wenn die Hälfte des Weins eine ungerade Maßzahl hat, aber alle Behältergrößen und ihre Füllstände gerade Zahlen sind, gibt es keine Lösung. (Diese Überlegung lässt sich auf die Teilbarkeit durch beliebige Zahlen verallgemeinern.)

Eine weitere Abkürzung lässt sich finden, wenn mehrere Behälter die gleiche Größe und zu Beginn die gleichen Füllstände haben. Dann braucht man zwischen diesen Behältern nicht zu unterscheiden: für jede gefundene Lösung gibt es dann eine weitere Lösung, in der die gleich großen Behälter gleichermaßen vertauscht werden.

1.6 Bewertungskriterien

- Das Verfahren zur Lösungssuche soll nachvollziehbar beschrieben sein, und es muss begründet sein, warum das Verfahren korrekte Ergebnisse liefert.
- Einfache Heuristiken, die bei den kleinen Beispielen keine Lösung finden, obwohl es eine gibt, sind nicht ausreichend. Das gleiche gilt für allzu aufwändige Ansätze. Insbesondere sollte erkannt werden, dass eine Befüllung, sofern einmal gefunden, nicht wieder betrachtet zu werden braucht. Heuristiken, die eine Lösung finden, obwohl es keine gibt, kommen wahrscheinlich nicht vor, da ja der Lösungspfad angegeben werden muss.
- Inkorrekte Ergebnisse wird eine Lösung insbesondere dann liefern, wenn das Umfüllen selbst falsch berechnet wurde.
- Der Lösungspfad (in der Aufgabenstellung „Plan“ genannt) soll übersichtlich ausgegeben sein. Es ist akzeptabel, wenn – wie in der Aufgabenstellung gefordert – nur die nötigen Umfüllungen angegeben sind. Sehr sinnvoll ist es aber, zusätzlich die aktuellen Füllzustände auszugeben. Dann kann viel besser nachvollzogen werden, ob die einzelnen Befüllungen zu den Behältergrößen passen und ob eine Befüllung in die folgende Befüllung überführbar ist.
- Die Ausgabe des Programms für die Pflichtbeispiele von Aufgabenblatt und BwInf muss dokumentiert sein. Weitere Beispiele, insbesondere größere, sind wünschenswert, aber nicht zwingend erforderlich.

Aufgabe 2: Mobile

2.1 Teil 1: Textuelle Darstellung – Lösungsidee

Um die Balance eines Balkens zu prüfen, an dem k Strukturen (Figuren oder andere Balken) hängen, benötigt man $2k$ Angaben: Die Gewichte w_i und die Positionen a_i der am Balken hängenden Strukturen ($1 \leq i \leq k$). Dabei werden die a_i ausgedrückt als horizontale Koordinaten bezogen auf den Aufhängepunkt des Balkens als Nullpunkt; also bedeutet bei der üblichen Orientierung der horizontalen Koordinatenachse ein negatives a_i, dass sich die Struktur links vom Aufhängepunkt befindet; Strukturen rechts vom Aufhängepunkt haben positives a_i.

Der Balken ist genau dann im Gleichgewicht, wenn $\sum_{i=1}^{n} a_i w_i = 0$.

Wenn die i-te Struktur eine einzelne Figur ist, dann kennt man ihr Gewicht direkt. Wenn die i-te Struktur ein anderer Balken ist, dann ergibt sich ihr Gewicht als Summe der an diesem Balken hängenden tieferen Strukturen; in diesem Fall benötigt man für die Ermittlung von w_i also die Kenntnis des tieferen Balkens.

Im folgenden werden zwei Varianten beschrieben, wie man angehängte Balken angeben kann: Entweder man führt explizite Schlüssel für alle Balken ein, auf die man dann verweisen kann, oder man vereinbart eine Reihenfolge, in der die untereinander hängenden Balken aufgeschrieben werden.

Erste Variante

In der ersten Variante kann man einen Balken durch eine Textzeile darstellen, die den Schlüssel des Balkens und pro angehängter Struktur zwei weitere Angaben enthält: vor der Position a_i entweder bei Figuren das Figurgewicht w_i oder bei Balken den Schlüssel s_i des angehängten Balkens.

Das Beispielmobile des Aufgabenblatts würde dann bei Verwendung von Großbuchstaben als Schlüssel und von Leerzeichen zur Trennung der Angaben folgendermaßen dargestellt:

```
A B -4 2 -1 3 2 2 4
B 1 -2 2 1
```

Statt bei einem Balken anzugeben, welche anderen Balken an ihm hängen, könnte man umgekehrt bei jedem Balken auch angeben, an welchem Elternbalken und wo er dort hängt. Dazu kann man beispielsweise die Position am Elternbalken und dessen Schlüssel vor die übrigen Angaben schreiben, wobei der oberste Balken zwei Sonderzeichen vorangestellt bekommt. Wieder mit Großbuchstaben als Schlüsseln:

```
 _ _ A 2 -1 3 2 2 4
-4 A B 1 -2 2 1
```

Zweite Variante

In der zweiten Variante ist die Einführung von Schlüsseln für die Balken nicht erforderlich. Angehängte Figuren werden wieder durch die Angaben a_i und w_i charakterisiert. Bei angehängten Balken reicht es aus, neben a_i ein spezielles Zeichen anzugeben, das signalisiert, dass hier ein weiterer Balken hängt; welcher das ist, ist aus der verabredeten Reihenfolgenkonvention klar. Eine günstige Reihenfolge ist, die Balken so zu durchlaufen, wie man das von der Tiefensuche her kennt: Direkt nach einem gegebenen Balken B0 ist (falls vorhanden) der Balken B1 an der Reihe, der am weitesten links an ihm befestigt ist. Der von links gezählt nächste direkt unter B0 hängende Balken B2 kommt (falls vorhanden) jedoch erst an die Reihe, nachdem alle direkt oder indirekt unter B1 hängenden Balken aufgelistet sind.

Das Beispielmobile sieht in dieser Darstellung folgendermaßen aus, wenn als spezielles Zeichen zur Kennzeichnung von angehängten Balken 0 verwendet wird (eine Verwechslungsgefahr mit einem Gewicht besteht nicht, da Gewichte immer positive Zahlen sind):

```
0 -4 2 -1 3 2 2 4
1 -2 2 1
```

Komplexere Beispiele folgen weiter unten.

Eine sehr verwandte Möglichkeit besteht darin, die Folgebalken nicht durch neue Zeilen, sondern durch Klammerausdrücke darzustellen. Das Beispielmobile sieht dann so aus:

```
(1 -2 2 1) -4 2 -1 3 2 2 4
```

2.2 Teil 1: Textuelle Darstellung – Umsetzung

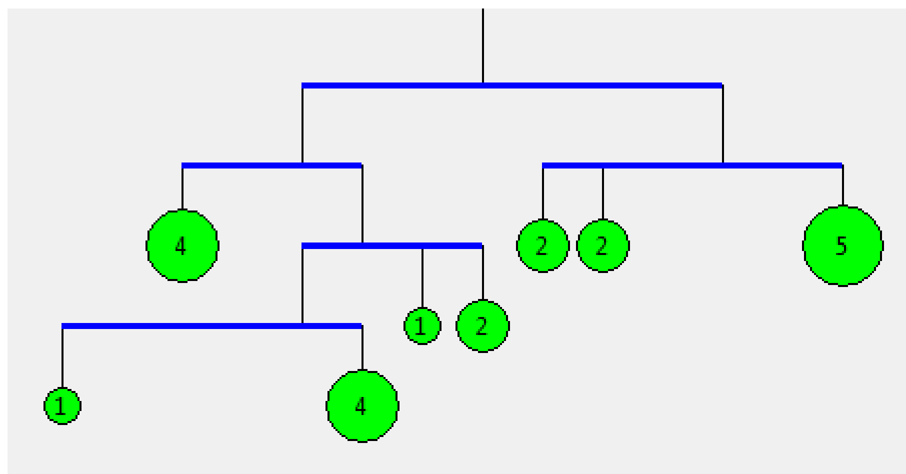
Die zweite Variante (die man auch als Notation in Tiefensuchen-Reihenfolge bezeichnen kann) erlaubt eine besonders effiziente Prüfung der Balanciertheit durch eine rekursiv aufgerufene Funktion `balken(textzeile)`, die im ausbalancierten Fall das Gewicht des durch `textzeile` repräsentierten Balkens zurückgibt und sonst -1. Der Pseudocode dieser Funktion:

```
funktion balken(textzeile):
    setze gesamt = 0
    setze balance = 0
    für alle zahlenpaare w, a in textzeile:
        wenn w = 0:
            w = balken(nächste textzeile)
            wenn w = -1
                beende mit Rückgabe w
        erhöhe gesamt um w
        erhöhe balance um a*w
    wenn balance = 0:
        beende mit Rückgabe gesamt
    sonst:
        beende mit Rückgabe -1
```

2.3 Teil 1: Textuelle Darstellung – Beispiele

Das folgende Beispiel zeigt die textuelle Darstellung eines balancierten Mobiles, welches darunter abgebildet ist. Wenn man die Funktion `balken` startend mit der ersten Textzeile anwendet, wird 21 als Gesamtgewicht des Mobiles zurückgegeben.

```
0 -3 0 4
4 -2 0 1
0 -1 1 1 2 2
1 -4 4 1
2 -3 2 -2 5 2
```



Falls man hingegen

```
0 -3 0 4
4 -2 0 1
0 -1 1 1 2 2
1 -3 3 1
2 -3 2 -2 5 2
```

verwendet, so ist die dritte Zeile nicht ausbalanciert, und es wird -1 zurückgegeben.

2.4 Teil 2: Mobiles konstruieren – Lösungsidee

Aus beliebigen Gewichten lässt sich ein gültiges Mobile bauen: Jeder Balken kann ausbalanciert werden, indem seine letzte Figur w_n an der durch die folgende Formel berechneten Position a_n aufgehängt wird:

$$a_n = \frac{-\sum_{i=1}^{n-1} a_i w_i}{w_n}$$

Man hat also bei der Konstruktion eines Mobiles so viel Freiheit, dass man bis auf die letzte Figur jedes Balkens die Gewichte zufällig verteilen kann, wobei hier noch die Einschränkung beachtet werden muss, dass nicht mehr als vier Strukturen an einen Balken hängen dürfen.

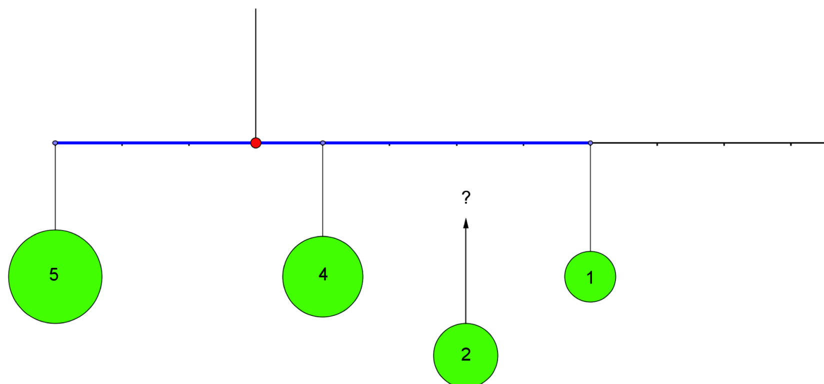


Abbildung 3: Der Aufhängepunkt des Balkens wird als gegeben betrachtet. Wenn an ihm k Gewichte angehängt werden sollen, kann man die Positionen von $k-1$ Gewichten beliebig (zufällig) wählen und anschließend die Position des k-ten Gewichtes so berechnen, dass der Balken im Gleichgewicht ist.

Wenn man keine weiteren Einschränkungen treffen möchte, kann man das Mobile zufällig von unten nach oben konstruieren: Das heißt, die Anzahl der an einem Balken zu befestigenden Strukturen (1 bis 4), die Strukturen selbst und fast alle Abstände wählt man zufällig.

Es gibt viele Varianten, Möglichkeiten und Einschränkungen, mit denen man „uninteressante“ oder „unschöne“ Fälle ausschließen kann, wie Balken mit nur einer Struktur oder mit extrem ungleichmäßig verteilten Strukturen. Letzteres ist auch ein Aspekt, der bei der grafischen Darstellung wichtig ist. Eine ungleichmäßige Verteilung lässt sich z. B. dadurch vermeiden, dass man möglichst ähnliches Gewicht auf beiden Seiten der Aufhängepunkte verteilt.

Man kann alternativ auch zunächst alle n Strukturen, die an einem Balken hängen sollen, an beliebigen Positionen x_i befestigen und anschließend den Schwerpunkt X des Balkens so bestimmen:

$$X = \frac{\sum_{i=1}^{n} x_i w_i}{\sum_{i=1}^{n} w_i}$$

Wenn man diesen als Aufhängepunkt des Balkens wählt, besteht Gleichgewicht. Die x_i können dann leicht mit $a_i = x_i - X$ in Abstände vom Aufhängepunkt umgerechnet werden.

Wenn man die x_i dann nicht zufällig wählt, sondern beispielsweise äquidistant, hat man von vornherein eine ungleichmäßige Verteilung ausgeschlossen. Für die grafische Darstellung ist noch zu erwähnen, dass sich in der zweidimensionalen Zeichnung möglicherweise Überlappungen ergeben. Bei realen dreidimensionalen Mobiles stört dies weniger.

2.5 Teil 2: Mobiles konstruieren – Umsetzung

Die oben beschriebene zufällige Konstruktion eines neuen Balkens kann durch folgende Funktion `hängen(freie_strukturen)` umgesetzt werden, wobei `freie_strukturen` die noch nicht aufgehängten Teile sind; die Funktion hat zwei Rückgabewerte: den erzeugten Balken als neue Struktur und die noch immer nicht aufgehängten Teile:

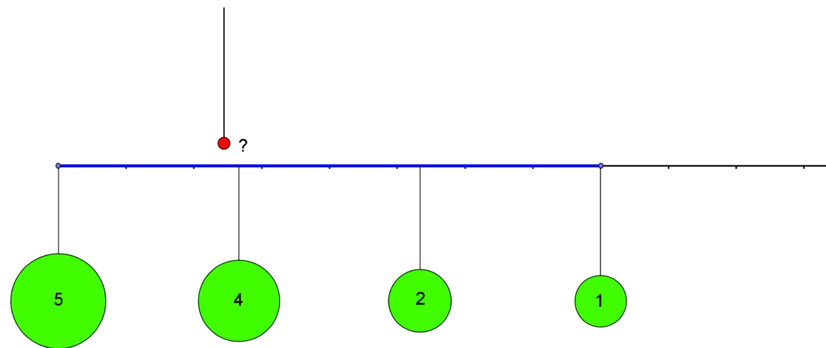


Abbildung 4: Der Aufhängepunkt ist zunächst nicht bekannt. Es werden erst alle k Gewichte beliebig (z.B. äquidistant oder zufällig) am Balken aufgehängt. Dann wird der Schwerpunkt der Anordnung berechnet. Wenn man den Balken dort aufhängt, ist er im Gleichgewicht.

```
funktion hängen(freie_strukturen):
    setze anzahl = ganze Zufallszahl (2 bis min(4, länge(
       freie_strukturen))
    setze auswahl = Zufallsauswahl von anzahl Elementen aus
       freie_strukturen
    setze rest = freie_strukturen ohne auswahl
    setze balance = 0
    für alle Strukturen stru bis auf letzte von auswahl:
        setze abstand(stru) = Zufallszahl im Bereich der Balkenlänge
        erhöhe balance um abstand(stru)*gewicht(stru)
    setze stru = letzte von auswahl
    setze abstand(stru) = -balance/gewicht(stru)
    setze neue_struktur = {gewicht(stru), abstand(stru)|alle stru aus
        auswahl}
    beende mit Rückgabe neue_struktur, rest
```

Anfangs sind `freie_strukturen` die n vorgegebenen Gewichte. Die Funktion `hängen` wird solange aufgerufen, bis der zweite Rückgabewert von `hängen` leer ist. Dabei muss ein neu konstruierter Balken in den zur Verfügung stehenden Teilen ergänzt werden, damit dieser auch aufgehängt wird:

```
setze offen = gegebene Figuren
wenn länge(offen) = 1:
    setze mobile = Trivialer Balken mit gegebener Figur im Abstand 0
    ende
setze mobile = leer
solange länge(offen) > 1:
    setze balken_neu, offen = hängen(offen)
    füge zu offen balken_neu hinzu
    füge zu mobile balken_neu hinzu
```

2.6 Teil 2: Mobiles konstruieren – Beispiele

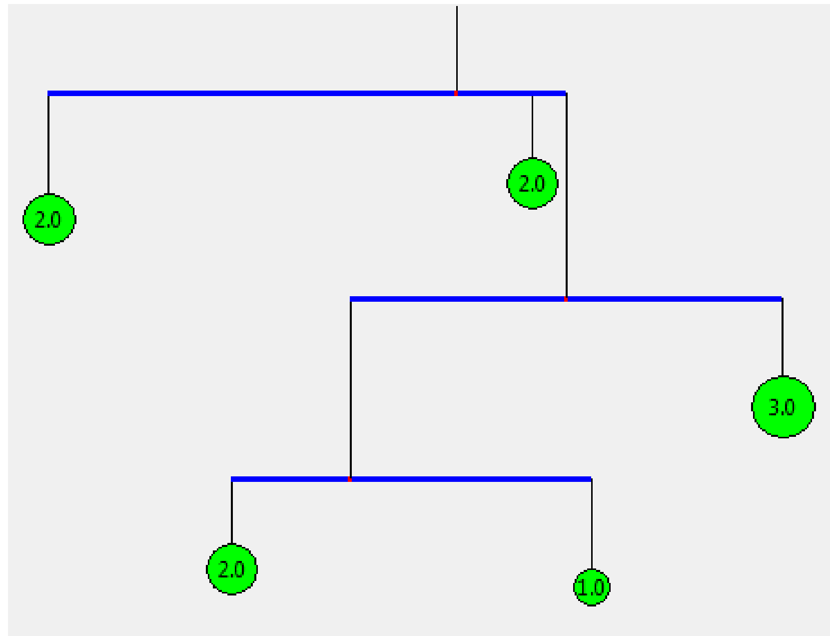


Abbildung 5: Mobile aus Beispiel 1

Beispiel 1 Gibt man als Eingabegewichte

```
1 2 2 2 3
```

wie in der Aufgabenstellung ein, so wird nach obigem Prinzip z. B. folgendes zufällige Mobile konstruiert (vgl. Abbildung 5):

```
2.0 2.10401276275 0 3.07399680931 2.0 -11.3260031907
0 -6.0 3.0 6.0
2.0 -3.33333333333 1.0 6.66666666667
```

Beispiel 2 Für Eingabegewichte

```
1 1 2 2 2 4 4 5
```

wird z. B das folgende Mobile zufällig konstruiert (vgl. Abbildung 6):

```
0 -0.685714285714 1.0 13.7142857143
2.0 6.97891698537 4.0 4.01643756419 1.0 -5.02108301463 0
   -1.92326932407
2.0 -1.21110987342 4.0 1.33236330512 5.0 -3.27246192481 2.0
   6.72753807519
```

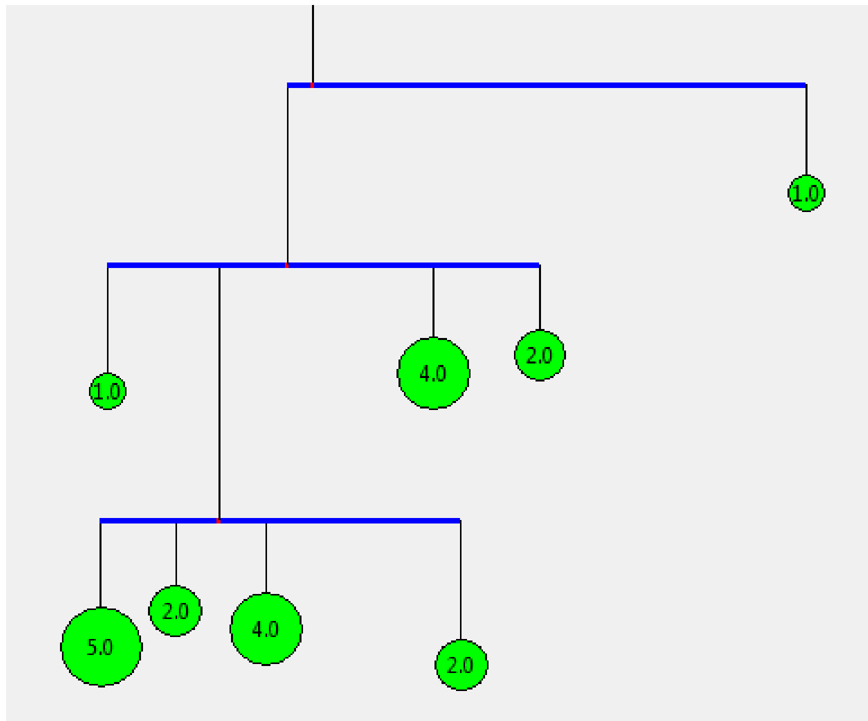


Abbildung 6: Mobile aus Beispiel 2

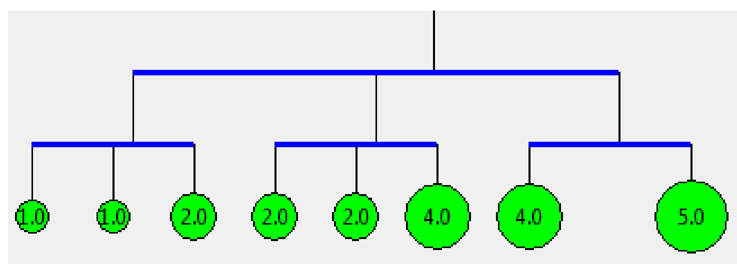


Abbildung 7: Mobile aus Beispiel 3

Beispiel 3 Wohingegen eine äquidistante Vorgehensweise für dieselbe Eingabe z. B. folgendes Mobile liefert (vgl. Abbildung 7):

```
0 -5.57142857143 0 -1.07142857143 0 3.42857142857
1.0 -1.875 1.0 -0.375 2.0 1.125
2.0 -1.875 2.0 -0.375 4.0 1.125
4.0 -1.66666666667 5.0 1.33333333333
```

2.7 Bewertungskriterien

- Die textuelle Darstellung und das Verfahren zur Mobilekonstruktion sollen nachvollziehbar beschrieben und ihre Wahl soll gut begründet sein.
- Für die textuelle Darstellung ist wichtig, dass ein Programm sie einigermaßen leicht einlesen kann. Eine bildhafte Darstellung per ASCII-Art besteht zwar aus Textzeichen, ist aber eben bildhaft und nicht textuell, und es ist eher schwierig, die für die Überprüfung der Balanciertheit nötigen Informationen daraus auszulesen. Die Darstellung muss nicht „frei erfunden" sein; Darstellungen auf der Basis von XML, JSON o.Ä. sind absolut in Ordnung.
- Die Aufgabenstellung fordert, dass die textuelle Darstellung eine Überprüfung der Balanciertheit erlauben soll. Das legt nahe, dann auch zu erläutern, wie diese Überprüfung erfolgt. Eine solche Erläuterung fehlt in vielen Einsendungen, insbesondere wenn das Verfahren zur Mobile-Konstruktion in Teil 2 nur balancierte Mobiles liefert. Das wird akzeptiert; erwartet wird aber dann in Teil 2 umso mehr, dass die Balanciertheit der konstruierten Mobiles gut begründet wird.
- Die für gegebene Gewichte konstruierten Mobiles müssen balanciert sein. Außerdem soll die Grundbedingung erfüllt sein, dass maximal vier Strukturen an einem Balken befestigt sind.
- Einigermaßen ausgewogen aussehende Mobiles sind schön, aber auch weniger schöne Mobiles werden akzeptiert, wenn die obigen Bedingungen erfüllt sind. Selbst die ganz einfach zu konstruierenden „binären" Mobiles (genau zwei Strukturen pro Balken) sind in Ordnung, solange sie nicht völlig einseitig mit Gewichten behängt sind.
- Sowohl für die textuelle Darstellung als auch für die Konstruktion balancierter Mobiles sollten jeweils mindestens drei Beispiele angegeben sein. Die textuelle Darstellung soll auch für das Beispiel-Mobile vom Aufgabenblatt angegeben werden. Zumindest für ein kleineres Beispiel sollte „vorgerechnet" sein, dass das konstruierte Mobile balanciert ist.
- Wurde eine grafische Darstellung des Mobiles realisiert und damit die freiwillige Zusatzaufgabe bearbeitet, wird dies zwar nicht mit Pluspunkten, aber mit einem '+' belohnt.

Aufgabe 3: Buffet-Lotterie

3.1 Lösungsidee

Eine naheliegende Lösungsidee ist es, die Buffet-Lotterie zu simulieren.

Wir starten mit n ($= 28$) Teilnehmern. Weil diese schon wissen, dass sie eine Weile mit dem Bestimmen der Buffetreihenfolge beschäftigt sein werden, setzen sie sich in einen Stuhlkreis. Die Stühle nummerieren sie von 0 bis $n-1$, wobei das Geburtstagskind (nennen wir sie der Lesbarkeit halber Gina) sich auf den Stuhl mit der Nummer 0 setzt.

Geht ein Teilnehmer zum Buffet, so verlässt er seinen Stuhl, die Teilnehmer rechts von ihm rücken einen Platz auf und der Stuhl mit der Nummer $n-1$ wird aus dem Kreis entfernt.

Nun können wir uns merken, auf welchem Stuhl der Teilnehmer sitzt, der als letztes eine Silbe des Satzes (der k ($= 16$) Silben habe) gesagt hat. Nennen wir diese Nummer i ("Index"). Am Ende des Satzes ist i die Nummer des Stuhls, dessen Teilnehmer zum Buffet gehen kann. Es ist gleichzeitig (wegen des Aufrückens der verbleibenden Teilnehmer) auch die Stuhlnummer des Teilnehmers, der mit der ersten Silbe wieder beginnt, den Satz erneut aufzusagen. Ein Beispiel ist in Abbildung 8 dargestellt.

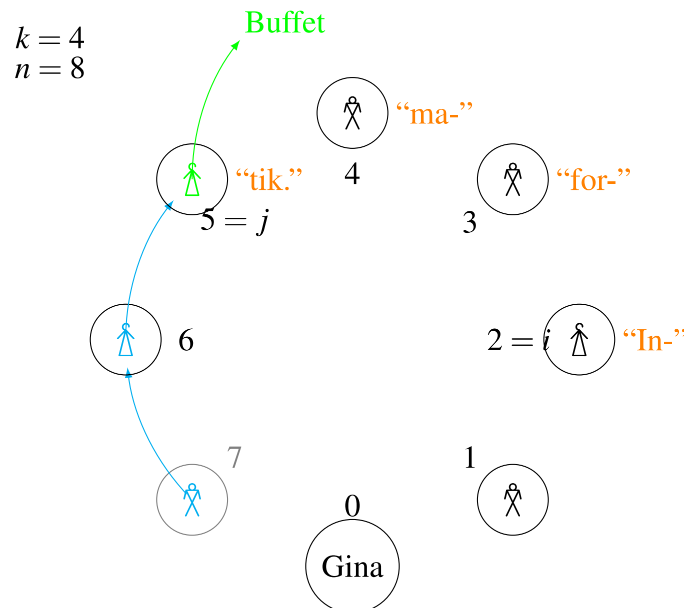


Abbildung 8: Veranschaulichung der Lösungsidee. Ein Satz mit nur vier Silben ("In-for-ma-tik.") wird einmal aufgesagt. Die Teilnehmerin auf Stuhl 5 darf als nächstes ans Buffet gehen, die Teilnehmer von Stuhl 6 und 7 rücken auf, Stuhl 7 wird entfernt.

Eine erste Simulation

Gehen wir zunächst davon aus, dass Gina immer nur genau eine Silbe des Satzes sagt, wenn sie an der Reihe ist. Dann können wir folgendermaßen berechnen, als wievielte sie ans Buffet gehen darf:

Beginne mit $i = 0$, da Gina die allererste Silbe sagt. Solange Gina noch nicht zum Buffet gehen durfte: Erhöhe i um $k-1$. Falls nötig, bilde den Rest modulo n (nach dem Stuhl $n-1$ kommt wieder der Stuhl 0, weil sie kreisförmig angeordnet sind) – das ist die Stuhlnummer des Teilnehmers, der als nächstes ans Buffet gehen darf. Verringere n um 1, da ja der letzte Stuhl entfernt wird.

Sobald wieder $i = 0$ ist, darf Gina ans Buffet gehen. Sitzen anschließend noch m der n Teilnehmer im Stuhlkreis, so bedeutet dies, dass Gina als $(n-m)$-te Teilnehmerin ans Buffet gehen darf; diese Zahl nennen wir p ("Platz in der Buffetreihenfolge").

Nutze den Geburtstagsvorteil

Nun darf Gina aber immer, wenn sie an der Reihe ist, auch zwei statt nur einer Silbe sagen. Dies will sie dazu nutzen, p zu minimieren. Wir müssen diesen Vorteil in unserer Simulation berücksichtigen. Zwar könnten wir jede Möglichkeit simulieren, wie Gina sich entscheiden könnte, das wären aber exponentiell viele: immer, wenn sie an die Reihe kommt, müssten wir zwei Möglichkeiten weiter simulieren.

Glücklicherweise geht es auch leichter. Welche Auswirkungen hat es auf die Simulation, wenn Gina einmal eine Silbe mehr sagt? Anstatt des Teilnehmers auf Stuhl $i + (k-1) \mod n =: j$ sagt der Teilnehmer auf Stuhl $i + (k-2) \mod n = j-1$ die letzte Silbe des Satzes. Dabei ist es offensichtlich egal, in welcher der vorigen Runden Gina zwei Silben gesagt hat. Das heißt, durch d-maliges Doppelt-Sprechen verringert Gina den Index insgesamt um d. Gina hat also „gewonnen“, sobald es ihr möglich ist, den Index auf 0 zu verringern: nämlich wenn $i \leq d$.

Modifikation der Simulation

Wir merken uns zusätzlich zum momentanen Index i und der Anzahl n der verbleibenden Teilnehmer auch noch, wie häufig Gina bisher an der Reihe war – das ist die Anzahl c von Chancen, bei denen Gina zwei statt einer Silbe sagen könnte. Wenn in einer Iteration $i \leq c$ ist, kann Gina also bis dahin schon i mal doppelt-sprechen und sich somit in dieser Iteration ihren Platz in der Buffetreihenfolge sichern.

c ist am Anfang 1 und erhöht sich immer, wenn der Index größer wird als $n-1$. Es könnte sein, dass der Satz viel länger ist als n, z. B. wenn schon 23 Teilnehmer am Buffet sind und nur noch fünf verbleiben. Dann kommt jeder dieser verbleibenden Teilnehmer bis zu vier mal an die Reihe, um den Satz mit 16 Silben ein einziges Mal aufzusagen. Sei $j = i + (k-1)$ der neue Index vor Modulorechnung. Dann müssen wir $\lfloor \frac{j}{n} \rfloor$ mal n abziehen, um auf eine gültige Stuhlnummer zu kommen. Im Laufe eines Satzes erhöht sich c somit um $\lfloor \frac{j}{n} \rfloor$.

Nun können wir in jeder Iteration testen, ob Gina bereits häufig genug zwei Silben sagen kann, um selbst als nächste ans Buffet gehen zu dürfen: Ist $i \leq c$? In diesem Fall sollte Gina i mal zwei Silben statt einer sagen (wobei es egal ist, zu welchem Zeitpunkt sie dies tut), z.B. bei den ersten i Gelegenheiten.

Algorithmus 8 Simulationsalgorithmus

```
n ← anzahlTeilnehmer
i ← 0
c ← 1
while true do
    i ← i + anzahlSilben − 1
    if i ≥ n then                                  ▷ Gina kommt wieder an die Reihe.
        c ← c + ⌊i/n⌋
        i ← i mod n
    end if
    n ← n − 1
    if i ≤ c then                  ▷ Gina kann den Index weit genug verringern.
        p ← anzahlTeilnehmer − n
        return "Gina bekommt Platz p, wenn sie i mal zwei Silben sagt."
    end if
end while
```

3.2 Algorithmus

Laufzeit

Jede Iteration der `while`-Schleife benötigt konstante Zeit. Es ist n die Anzahl der verbleibenden Teilnehmer, also nie negativ und nie größer als die Gesamtzahl der Teilnehmer. n wird aber in jeder Iteration um eins verringert, d. h. nach spätestens n Iterationen terminiert der Algorithmus. Insgesamt ist die Laufzeit also linear in der Anzahl der Teilnehmer.

Korrektheit

Ist irgendwann $i \leq c$, so gibt der Algorithmus offensichtlich eine mögliche Lösung aus. Das berechnete p ist auch tatsächlich optimal: Für alle früheren Iterationen ist $i > c$, Gina kann aber pro Runde nur eine Silbe mehr sagen, i also nur um bis zu c verringern.

3.3 Beispiele

- 28 Personen, 16 Silben. (Werte aus der Aufgabenstellung)
  Gina bekommt Platz 8, wenn sie 2 mal zwei Silben sagt.
- 16 Personen, 16 Silben.
  Gina bekommt Platz 2, wenn sie 0 mal zwei Silben sagt.
- 15 Personen, 16 Silben.
  Gina bekommt Platz 1, wenn sie 0 mal zwei Silben sagt.
- 55 Personen, 16 Silben.
  Gina bekommt Platz 10, wenn sie 3 mal zwei Silben sagt.
- 28 Personen, 32 Silben.
  Gina bekommt Platz 5, wenn sie 1 mal zwei Silben sagt.

- 10 Personen, 8 Silben.
  Gina bekommt Platz 4, wenn sie 4 mal zwei Silben sagt.
- 5 Personen, 17 Silben.
  Gina bekommt Platz 1, wenn sie 1 mal zwei Silben sagt.
- 280 Personen, 19 Silben.
  Gina bekommt Platz 78, wenn sie 5 mal zwei Silben sagt.
- 9 Personen, 312 Silben.
  Gina bekommt Platz 1, wenn sie 5 mal zwei Silben sagt.

3.4 Sinnvolle Ergänzung

Für Gina ist es egal, in welchen i der ersten c Runden sie zwei Silben statt einer sagt, für die anderen Teilnehmer jedoch nicht. Vielleicht kann sie erreichen, dass ihre besten Freunde auch möglichst gute Plätze in der Buffetreihenfolge bekommen?

3.5 Bewertungskriterien

- Das gewählte Verfahren soll nachvollziehbar beschrieben und seine Wahl begründet sein.
- Die Aufgabenstellung fordert zwar nur, dass die Lösung mit verschiedenen Teilnehmerzahlen zurecht kommt. Das gewählte Verfahren (aber nicht zwingend auch die Implementierung) sollte aber grundsätzlich auch mit anderen Sätzen bzw. Silbenzahlen als 16 funktionieren.
- Ein Brute-Force-Ansatz, wie oben kurz erläutert, kann im Extremfall schon für die im Beispiel genannten 28 Teilnehmer zu ineffizient sein und ist dann nicht ausreichend.
- Das Verfahren sollte korrekte Ergebnisse liefern.
- Es sollten genug aussagekräftige Beispiele angegeben sein. Ausgegeben werden sollte zum einen, wie oft Gina 2 Silben sagen muss, und zum anderen, welchen Platz in der Buffetreihenfolge sie damit erreicht (also in welcher Runde sie ans Buffet kommt). In vielen Einsendungen wird der Platz leider nicht angegeben. Das wird akzeptiert, da die Aufgabenstellung es nicht ausdrücklich verlangt.

Aufgabe 4: Alphametiken

Für diese Aufgabe untersuchen wir Gleichungen der Form

$$\mathtt{SUCHEN} - \mathtt{MACHT} = \mathtt{SPASS}$$

wobei jeder Buchstabe genau eine Ziffer ersetzt. Zwei unterschiedliche Ziffern werden immer durch zwei unterschiedliche Buchstaben repräsentiert und zwei unterschiedliche Buchstaben repräsentieren auch immer zwei unterschiedliche Ziffern. Diese Gleichungen, in denen auf der linken Seite beliebig viele Wörter mit den Operatoren +, −, ⋆ und / verknüpft werden und auf der rechten Seite nur ein einzelnes Wort steht, werden *Alphametiken* genannt.

4.1 Normalisierung von Alphametiken

Alphametiken sind Gleichungen, in denen jeder Buchstabe so durch eine Ziffer ersetzt werden kann, dass die Gleichung erfüllt ist. Zur besseren Übersicht können die Buchstaben konsistent in x_i umbenannt werden, wobei x_i der $(i+1)$-te unterschiedliche Buchstabe ist. Wir beginnen unsere Zählung also bei 0. Gleichzeitig sollte deutlich gemacht werden, dass das Hintereinanderschreiben der Buchstaben keine Multiplikation bedeutet, sondern dem Hintereinanderschreiben von Ziffern im Stellenwertsystem entspricht. Dazu wird der Ausdruck in Klammern gesetzt und in den Index eine 10 geschrieben. Aus

$$\mathtt{SEND} + \mathtt{MORE} = \mathtt{MONEY}$$

wird damit

$$(x_0x_1x_2x_3)_{10} + (x_4x_5x_6x_1)_{10} = (x_4x_5x_2x_1x_7)_{10}$$

Es wurden die in Tabelle 1 aufgelisteten Substitutionen vorgenommen.

Suche	Ersetze	Suche	Ersetze	Suche	Ersetze	Suche	Ersetze
S	x_0	E	x_1	N	x_2	D	x_3
M	x_4	O	x_5	R	x_6	Y	x_7

Tabelle 1: Substitutionen für die Alphametik „SEND+MORE=MONEY“.

Außerdem kann die rechte Seite der Gleichung von beiden Seiten subtrahiert werden:

$$(x_0x_1x_2x_3)_{10} + (x_4x_5x_6x_1)_{10} = (x_4x_5x_2x_1x_7)_{10} \quad (6)$$

$$\Leftrightarrow \underbrace{(x_0x_1x_2x_3)_{10}}_{=:A} + \underbrace{(x_4x_5x_6x_1)_{10}}_{=:B} - \underbrace{(x_4x_5x_2x_1x_7)_{10}}_{=:C} = 0 \quad (7)$$

4.2 Eine Alphametik lösen

Brute Force

An der neuen Schreibweise kann direkt abgelesen werden, dass es 8 Variablen gibt. Ein Brute-Force-Ansatz, der für jede Variable jede Ziffer ausprobiert, benötigt $10^8 = 100\,000\,000$ Versuche. Da jedoch zwei verschiedene Buchstaben auch zwei verschiedene Ziffern repräsentieren,

gibt es für die erste Variable 10 Möglichkeiten, für die zweite 9, für die dritte nur noch 8 usw. Insgesamt sind es also bei 10 Variablen $10! = 3\,628\,800 \approx 3{,}6 \cdot 10^6$ Möglichkeiten. Bei 8 Variablen gibt es daher $10 \cdot 9 \cdot 8 \cdot \ldots \cdot 3 = \frac{10!}{(10-8)!} \approx 1{,}8 \cdot 10^6$ zu überprüfende Möglichkeiten.

Es ist noch wichtig anzumerken, dass Zahlen nicht mit 0 beginnen. Es gilt also insbesondere für das Beispiel

$$x_0 \neq 0 \text{ und } x_4 \neq 0$$

Damit sind es nur noch $\frac{9 \cdot 8 \cdot 8!}{(10-8)!} = 1\,451\,520$ Möglichkeiten.

Auswertung

Jede mögliche Buchstaben-Ziffern-Belegung muss nun ausgewertet werden; d.h. es ist zu prüfen, ob nach dem Ersetzen der Buchstaben durch die gewählten Ziffern die Gleichung mathematisch erfüllt ist. Für einzelne Beispiele kann ein Algorithmus zur Auswertung ganz direkt implementiert werden. Man spricht hier von *Hardcoden*, weil diese Lösung nicht auf andere Beispiele ohne manuelle Arbeit übertragbar ist.

Eval Einige Programmiersprachen bieten die Möglichkeit, Zeichenketten zur Laufzeit als Code auszuwerten (engl. *evaluate*). Die einfachste Möglichkeit zur Auswertung besteht dann darin die Gleichung als Zeichenkette zu belassen, das `=` durch ein `-` zu ersetzen und eine `eval`-Funktion zu nutzen. Wenn das Ergebnis `0` ist, war die Belegung gültig. Diese Lösung ist jedoch häufig nicht wünschenswert. Zum einen ist sie nicht immer verfügbar (wie z. B. in C), zum anderen können Benutzer so jeden beliebigen Code ausführen. Dies kann zum Verhängnis werden, wenn die Benutzer Schadcode oder einfach eine falsche Zeichenkette eingeben.

Im allgemeinen kann die Auswertung einer Alphametik für eine gegebene Buchstaben-Ziffern-Belegung in zwei Schritten erledigt werden:

- Berechnung des Wertes eines Wortes in einer Alphametik
- Auswertung der gesamten Gleichung der Alphametik

Wort-Auswertung

Zur Auswertung eines einzelnen Wortes kann die die Belegung der Variablen $x_1, \ldots, x_{10}$ mit den Ziffern $0, \ldots, 9$ in einem Array B der Länge 10 gespeichert werden. Das *wort* kann dann als Liste repräsentiert werden, wobei die Elemente der Liste nach dem Stellenwertsystem geordnet sind. Also ist das erste Element von der Liste *wort* der letzte Buchstabe im Wort. Da die Buchstaben durch x_j ersetzt wurden, kann die Buchstaben-Zahl-Belegung jedes Wort also als eine Liste von Indices j von B repräsentiert werden.

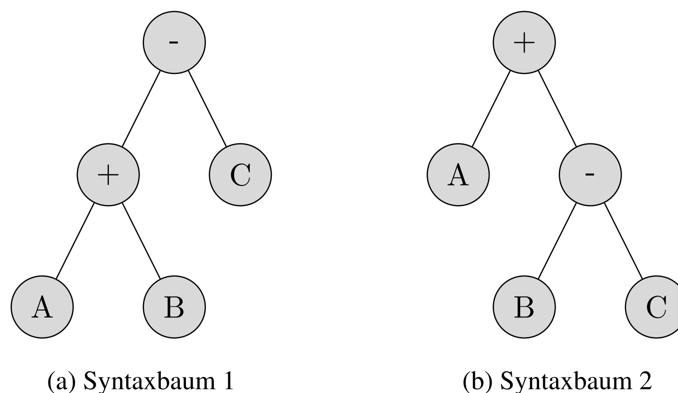
Algorithmus 9 zeigt in Pseudocode, wie die Auswertung eines Wortes mit einer gegebenen Belegung B funktioniert.

Algorithmus 9 Auswertung eines Wortes einer Alphametik mit gegebener Belegung B

```
B ← [1,5,3,7,2,9,4,6,8,0]                          ▷ x_0 = 1, x_1 = 5, ..., x_9 = 0
wort ← [3,2,1,0]                                   ▷ SEND = (x_0 x_1 x_2 x_3)_10
ergebnis ← 0
for stelle in 0,...,LEN(wort) − 1 do
    ergebnis ← ergebnis + 10^stelle · B[wort[stelle]]
end for
```

Auswertung der Gleichung: Syntaxbäume

Syntaxbäume sind in unserem Fall binäre Bäume, deren innere Knoten Operatoren und deren äußere Knoten Zahlen sind. Zwei korrekte Syntaxbäume für Gleichung (7) sind in Abb. 9 dargestellt.



(a) Syntaxbaum 1 (b) Syntaxbaum 2

Abbildung 9: Zwei mögliche Syntaxbäume für Gleichung (7). Die Gleichung wird von den Blättern aus evaluiert. In Syntaxbaum 1 wird also $(A+B)-C$ gerechnet, in Syntaxbaum 2 hingegen $A+(B-C)$.

Syntaxbäume benötigen keine Klammern zur Auswertung. Die Reihenfolge der Auswertung ist beliebig, jedoch benötigen die Operatoren zur Auswertung Zahlen. Daher wird von den Blättern ausgehend zur Wurzel hoch ausgewertet.

Der Syntaxbaum kann folgendermaßen erstellt werden:

- Der Wurzelknoten ist `-`, und auf der rechten Seite ist ein Blatt mit der rechten Seite der Gleichung.
- Auf der linken Seite wird nach `+` und `-` gesucht. Wird eines dieser Zeichen gefunden, so wird ein Kind-Knoten hinzugefügt und die Zeichenkette an dieser Stelle geteilt. Der Algorithmus zum Aufbau des Syntaxbaumes wird mit beiden Teilzeichenketten erneut aufgerufen.
- Wenn keine `+` und `-` mehr vorhanden sind, werden `*` und `/` gesucht und der Baum weiter gebaut. Gleichzeitig wird die Zeichenkette immer kleiner.
- Nun sollten nur noch Wörter übrig sein. Diese werden in die Blätter geschrieben.

Im Extremfall müssen so zwar immer noch knapp drei Millionen Buchstaben-Zahl-Zuordnungen für eine Alphametik ausprobiert werden, aber die Auswertung kann schnell durchgeführt werden, und sie muss nur für einzelne Alphametiken durchgeführt werden.

4.3 Erstellen von Zahl-Alphametiken

Es gibt sehr viele mögliche Alphametiken. Allerdings kann die Aufgabe deutlich vereinfacht werden, wenn nicht alle Möglichkeiten betrachtet werden. Da die Division zu Rundungsproblemen führen könnte, wird im Folgenden auf sie verzichtet. Auch wurden die Wörter auf deutsche Zahlwörter beschränkt. Außerdem wurde die Aufgabe so eingeschränkt, dass die erzeugten Alphametiken sowohl korrekte Gleichungen in ihrer Eigenschaft als Alphametik darstellen sollen, als auch in ihrer Eigenschaft als Zahlwörter. Daher ist die Alphametik

$$\texttt{EINS} = \texttt{DREI}$$

keine gültige Lösung, selbst wenn eine passende Belegung gefunden wird.

Im Folgenden wird eine Alphametik mit den gewünschten Eigenschaften eine *Zahl-Alphametik* genannt.

Auswahl der Zahlen

Eine wichtige Frage bei der Erzeugung einer Zahl-Alphametik ist die Wahl der Zahlen. Durch die Betrachtung der Binär-Schreibweise einer Laufvariable i kann die Menge bestimmt werden. So gibt $i = 11$ an, dass die Zahlen 1, 2 und 4 benutzt werden, da

$$(11)_{10} = (8+2+1)_{10} = (1011)_2$$

gilt. Die nächste Menge wäre $i = 12$ mit den Zahlen 3 und 4:

$$(12)_{10} = (8+4)_{10} = (1100)_2$$

Auf diese Art wird Schritt für Schritt jede endliche Teilmenge der natürlichen Zahlen erzeugt. Alles was gespeichert werden muss ist die Laufvariable i.

Sobald die Menge der zu betrachtenden Zahlen bestimmt wurde, wird die Zahl gewählt, die rechts stehen soll. Dann müssen passende Operatoren gefunden werden, sodass die Zahlwörter auf der linken Seite durch Operationen verknüpft das Wort der rechten Seite ergeben.

Eine Kurzschreibweise

Auf der linken Seite einer Gleichung kann man beliebig viele Zahlen haben. Um Platz zu sparen wollen wir eine Kurzschreibweise einführen. Die Alphametik

$$\texttt{EINS} + \texttt{EINS} + \texttt{EINS} + \texttt{EINS} = \texttt{VIER}$$

kann auch durch

$$4 \cdot \texttt{EINS} = \texttt{VIER}$$

ausgedrückt werden.

Der größte gemeinsame Teiler

Der *größte gemeinsame Teiler* (kurz: ggT) zweier natürlicher Zahlen a und b kann mit dem euklidischen Algorithmus berechnet werden. Man macht sich dabei die Eigenschaften des ggT zunutze, dass

$$\text{ggT}(a,b) = \text{ggT}(a-b,b) \text{ für alle } a,b \in \mathbb{N}$$

gilt. Daher kann durch wiederholte Subtraktion der ggT berechnet werden. Es gibt also insbesondere zwei ganze Zahlen x_1 und x_2 sodass:

$$x_1 \cdot a + x_2 \cdot b = \text{ggT}(a,b)$$

Mit dem *erweiterten euklidischen Algorithmus* (EEA) können x_1 und x_2 bestimmt werden.

Der ggT von vielen Zahlen kann durch wiederholtes Anwenden des ggT auf einzelne Zahlen berechnet werden. Insbesondere kann der EEA auch von mehr als zwei Zahlen die Koeffizienten x_i berechnen.

Beim EEA merkt man sich, wie häufig welche Zahlen subtrahiert wurden, und rechnet das Ergebnis zurück. Soll beispielsweise der ggT von 18 und 30 bestimmt werden, geht man wie folgt vor:

$$30 = 1 \cdot 18 + 12 \tag{8}$$
$$18 = 1 \cdot 12 + 6 \tag{9}$$
$$12 = 2 \cdot 6 + 0 \tag{10}$$

In Gleichung (10) sieht man, dass der EEA beendet ist, da 0 als Rest addiert wurde. Nun wird von Gleichung (9) aus zurück gerechnet:

$$6 = 18 - 1 \cdot 12 \tag{11}$$
$$= 18 - 1 \cdot (30 - 1 \cdot 18) \tag{12}$$
$$= (-1) \cdot 30 + 2 \cdot 18 \tag{13}$$

Die gesuchten Konstanten sind also $x_1 = -1$ und $x_2 = 2$. Es gilt:

$$\text{ggT}(30,18) = x_1 \cdot 30 + x_2 \cdot 18 = (-1) \cdot 30 + 2 \cdot 18 = 6$$

Insbesondere ist 6 die kleinste positive Zahl, die Ausdrücke der Form

$$x_1 \cdot 30 + x_2 \cdot 18 \text{ mit } x_1, x_2 \in \mathbb{Z}$$

als Wert annehmen können.

Wenn nun auf der linken Seite einer Alphametik zwei oder mehr Terme stehen, dann kann mithilfe des EEA der kleinste Ausdruck berechnet werden, der durch wiederholte Addition bzw. Subtraktion erzeugt werden kann. Kennt man erst diese Konstanten, so kann man den Gesamtausdruck auf der linken Gleichungsseite mit einer ganzzahligen Konstante multiplizieren, um das Wort auf der rechten Gleichungsseite zu erhalten. Wenn der ggT der Zahlwörter der linken Seite jedoch kein Teiler des Zahlwortes der rechten Seite ist, dann ist die Alphametik nicht durch ausschließliche Verwendung der Operatoren +, − und · modifizierbar und eine weitere Zahlwort-Kombination sollte ausprobiert werden.

Zusammenstellen der Puzzlestücke

Die beschriebenen Ideen können in einem Algorithmus zusammengebaut werden, der in der Lage ist, in wenigen Sekunden Laufzeit Alphametiken von deutlich mehr als 20 Buchstaben zu erzeugen. Eine Skizze dieses Algorithmus in Pseudocode sieht wie folgt aus:

Algorithmus 10 Generiere Zahl-Alphametiken

```
function GENERIERE
    for i ← 1; True; i ← i + 1 do
        zahlen ← ERZEUGEZAHLENMENGEAUSBINAERSCHREIBWEISE(i)
        w ← ERZEUGEZAHLWOERTER(zahlen)
        if UNTERSCHIEDLICHEBUCHSTABEN(w) > 10 then
            continue
        end if
        for rechts in zahlen do
            links ← zahlen \ { rechts }
            coeffizienten, ggT ← EEA(links)
            if rechts%ggT == 0 then
                mult_const ← rechts / ggT
                ▷ Die linke Seite mit mult_const ausmultiplizieren
                alpham ← ERZEUGEALPHAMETIK(links, rechts, mult_const)
                ▷ Nun ergeben die Zahlwörter eine korrekte Gleichung
                if BRUTEFORCELOESUNG(alpham) then
                    Ausgabe der Alphametik
                end if
            end if
        end for
    end for
end function
```

4.4 Lange Alphametiken

Es gibt viele Alphametiken mit mehr als 20 Buchstaben. Darunter sind:

- $3 \cdot (-2 \cdot ZWEI + FUENF) = DREI$ mit 43 Buchstaben
- $4 \cdot (-2 \cdot ZWEI + FUENF) = VIER$ mit 56 Buchstaben
- $7 \cdot (-2 \cdot ZWEI + FUENF) = SIEBEN$ mit 97 Buchstaben
- $9 \cdot ((-5 \cdot (-2 \cdot VIER + ZEHN) + ELF)) = NEUN$ mit 571 Buchstaben

4.5 Bewertungskriterien

- In beiden Teilaufgaben sollen die gewählten Verfahren nachvollziehbar beschrieben und ihre Wahl gut begründet sein.

- Ein Brute-Force-Ansatz ist für Teilaufgabe 1 in Ordnung, wenn die Möglichkeiten wie oben beschrieben eingeschränkt werden, also insbesondere – gemäß den Alphametik-Regeln – die eineindeutige Zuordnung zwischen Buchstaben und Ziffern berücksichtigt wird. Es wird akzeptiert, wenn die 0 als Belegung für alle Buchstaben erlaubt ist, Zahlen also auch mit 0 beginnen können; dies sollte aber sauber dokumentiert sein.
- Die problematische Laufzeit sollte insbesondere bzgl. der Generierung von Alphametiken angesprochen werden.
- Das Lösungsverfahren (Teilaufgabe 1) muss korrekt funktionieren. Die Lösung für das Pflichtbeispiel `SUCHEN - MACHT = SPASS` lautet `108347-95836=12511`, also: `S<-1,U<-0,C<-8,H<-3,E<-4,N<-7,M<-9,A<-5,T<-6,P<-2`.
  Das Verfahren kann auch auf `SEND + MORE = MONEY` getestet werden. Dann muss die Lösung `9567+1085=10652` gefunden werden. Wenn hierfür 605 Lösungen gefunden werden, ist nicht berücksichtigt, dass je zwei unterschiedliche Buchstaben auch unterschiedliche Werte repräsentieren müssen. 25 Lösungen werden gefunden, wenn Zahlen mit 0 beginnen dürfen.
- Für Aufgabenteil 2 (der natürlich nicht fehlen darf) sollte eine Zahl-Alphametik mit mehr als 20 Buchstaben gefunden werden. Einschränkungen z.B. bei der Menge der Operatoren sind akzeptabel, wenn sie begründet sind.
- Die Quelle der deutschen Zahlwörter muss angegeben werden oder es muss beschrieben werden, wie die Zahlwörter erzeugt wurden. Es sollte daraus hervorgehen, wie mit Umlauten (`FÜNF` oder `FUENF`, `DREIßIG` oder `DREISSIG`) umgegangen wird. Es ist OK, nur mit den Zahlwörtern von `EINS` bis `NEUN` oder ähnlich eingeschränkten Mengen zu arbeiten.
- Es sollte erkannt worden sein, dass Zahlwörtermengen mit mehr als 10 unterschiedlichen Buchstaben niemals eine Alphametik ergeben.
- Es müssen für beide Teile genügend Beispiele angegeben sein. Für Teil 1 muss die Lösung des Pflichtbeispiels dokumentiert sein. Für Teil 2 sollte für jedes Beispiel sowohl die Zahl-Alphametik als auch mindestens eine Lösung angegeben sein.

Aufgabe 5: Pong

Vorbemerkung: Im Folgenden beziehen wir uns gelegentlich auf die auf dem Turnierserver abgelegte *Spieldokumentation* zu „Pong“[8].

5.1 Das Spiel

Das Spiel *Pong* spielen 2 Spieler auf einem Spielfeld, welches aus quadratischen Zellen besteht. Dabei befinden sich horizontal 65 und vertikal 60 Felder neben- bzw. übereinander. Diese Zellen seien von der 0. Spalte und der 0. Zeile von unten links beginnend durchnummeriert.

Das Spielfeld sei nun in ein Koordinatensystem gelegt. Dabei sei der Zellenmittelpunkt eines Feldes $(i|j)$ an die Koordinaten $(i|j)$ in diesem Koordinatensystem gelegt. Das Spielfeld erstreckt sich also von einem Punkt $(0|0)$ bis zu einem Punkt $(64|59)$. Dem intuitiven Verständnis nach sei der Rand des Spielfelds mit $x = 64$ als *rechter Rand*, der Rand mit $x = 0$ als *linker Rand* bezeichnet.

Am linken und rechten Rand befinden sich nun je ein *Schläger*. Diese haben je eine rechteckige Form der Ausmaße 1×6 in x- bzw. y-Richtung. Dabei sind die Schläger je in 3 gleich große, übereinander angeordnete Abschnitte unterteilt, den *oberen*, den *mittleren* und den *unteren*. Zu jedem Zeitpunkt $t \in \{0, 1, \dots, 500\}$ kann jeder Spieler seinen Schläger um maximal 1 nach oben oder unten bewegen, sofern diese Operation den Schläger nicht aus dem Spielfeld bewegt; zum Zeitpunkt 0 befinden sich die Schläger an den Positionen $(0|27)$ bis $(0|32)$ bzw. $(64|27)$ bis $(64|32)$. Es ist jedem Spieler zu jedem Zeitpunkt bekannt, wo sich der gegnerische Schläger befindet.

Ein *Ball* $B(b_x|b_y), b_x, b_y \in \mathbb{N} \cup \{0\}$ bewegt sich nun in diesem Spielfeld. Die Koordinaten b_x, b_y stellen dabei jedoch nur Approximationen der wirklichen Position $B'(b'_x|b'_y)$ des Balles dar. Zu Beginn des Spiels bewegt sich der Ball mit der Geschwindigkeit $\vec{v}(0) = \vec{v}_0(t) = \binom{v_x(0)}{v_y(0)}, |\vec{v}_0(t)| = 1, t \in \mathbb{N} \cup \{0\}$[9]. Diese Geschwindigkeit $\vec{v}(t)$ vergrößert sich betragsmäßig mit ansteigender Zeit t, wie im Folgenden beschrieben.

Trifft der Ball auf eine obere oder untere Feldbegrenzung, gilt also zu einem Zeitpunkt $b_y = 0$ oder $b_y = 59$, so „prallt“ der Ball von dieser Begrenzung ab. Die Geschwindigkeit des Balles ändert sich dann von der ursprünglichen Geschwindigkeit $\binom{v_x(t)}{v_y(t)}$ in $\binom{1{,}05 \cdot v_x(t)}{-1{,}05 \cdot v_y(t)}$. Analog ändert sich die Geschwindigkeit beim Aufprallen auf einen mittleren Schlägerabschnitt von $\binom{v_x(t)}{v_y(t)}$ in $\binom{-1{,}05 \cdot v_x(t)}{1{,}05 \cdot v_y(t)}$. Trifft der Ball aber auf einen oberen oder unteren Schlägerabschnitt, so ändert sich seine Geschwindigkeit von $\binom{v_x(t)}{v_y(t)}$ in einen zufälligen Wert zwischen $\binom{-1{,}05 \cdot v_x(t)}{1{,}05 \cdot v_y(t)}$ und $\left|\binom{-1{,}05 \cdot v_x(t)}{1{,}05 \cdot v_y(t)}\right| \cdot \frac{\binom{1}{\tan 70°}}{\left|\binom{1}{\tan 70°}\right|}$ bzw. $-\left|\binom{-1{,}05 \cdot v_x(t)}{1{,}05 \cdot v_y(t)}\right| \cdot \frac{\binom{1}{\tan 70°}}{\left|\binom{1}{\tan 70°}\right|}$.

Gelangt der Ball im Verlauf des Spiels nun an die Stelle $x = 0$ oder $x = 64$, so hat der Spieler das Spiel verloren, dessen Schläger sich an dieser Stelle befindet. Ziel eines jeden Spielers ist es also offensichtlich, den Ball an den dem eigenen Schläger gegenüberliegenden Rand zu befördern.

[8] `http://di188.di.informatik.tu-darmstadt.de/static/Pong-Dokumentation.pdf`
[9] Bei der in der Spieldokumentation angegebenen Geschwindigkeit handelt es sich um einen Schreibfehler.

5.2 Lösungsidee

Wie in der Spieldokumentation beschrieben, sei nur eine Strategie für den linken Schläger betrachtet. Der eigene Schläger erstreckt sich also über die Felder mit den Koordinaten $(0|s_y + \delta), 0 \leq \delta \leq 5, 0 \leq s_y, s_y + 5 \leq \hat{y}$, der des Gegners analog über die Felder mit den Koordinaten $(64|s_y + \delta), 0 \leq \delta \leq 5, 0 \leq s_y, s_y + 5 \leq \hat{y}$

Treffen und zurückspielen

Unabhängig von der verfolgten Strategie, gilt es, zunächst die Aufschlagposition des Balles auf der eigenen Seite zu ermitteln. Es genügt nicht, den Schläger auf der y-Koordinate des Balles zu „halten". Das ist schon daran zu erkennen, dass sich der Ball von den Schlägern in einem Winkel von bis zu $\pm 70°$ wegbewegen kann. Der Ball kann sich also innerhalb einer Zeiteinheit um mehr als 1 Feld in y-Richtung bewegen; der Schläger kann dem Ball also gar nicht unbedingt folgen. Dieses Problem wird mit laufender Zeit noch größer, da sich der Betrag der Geschwindigkeit des Balles erhöht.

Zur Berechnung der Aufschlagposition könnte der Weg des Balles simuliert werden. Da der Ball von den oberen und unteren Begrenzungen jedoch im selben Winkel abprallt wie er auf diese aufschlägt, kann die Aufschlagposition nach den Sätzen der Elementargeometrie auch mit dem Schnittpunkt $S(1|s_y)$ der Parallelen zur y-Achse durch $x = 1$ und der Geraden $g : \vec{x} = \binom{b_x}{b_y} + r \cdot \vec{v}(t), r \in \mathbb{R}$ berechnet werden. Dabei müssen die Schnittpunktkoordinaten keine natürlichen Zahlen sein. Aufgrund der Ungenauigkeit der Position des Balles sollte die Aufschlagposition mehrfach berechnet werden, um eine möglichst hohe Genauigkeit zu erzielen.

Zur Berechnung dieses Schnittpunktes genügt es, speziell die Gerade g zu ermitteln, da eine Koordinate des Schnittpunktes ja bereits bekannt ist. Eine Gerade kann bekanntermaßen aus 2 verschiedenen Punkten ermittelt werden, es können also einzig Probleme auftreten, wenn der Ball gerade zwischen diesen Zeitpunkten von einer Wand abgeprallt ist. Auch andere Möglichkeiten zur Bestimmung der Geraden sind denkbar. So sind aus der Physik z.B. die Methode der *Ausgleichsgeraden* oder auch generell Methoden aus der Fehlerrechnung anwendbar.

Hat man nun den Schnittpunkt ermittelt, so kann die eigentliche Aufschlagposition nur noch an den y-Koordinaten $s_y \bmod 60$ oder $59 - (s_y \bmod 60)$ liegen. An welcher der beiden Stellen der Ball aufschlägt, hängt dabei nur von der Anzahl a der Abprallungen von einer oberen oder unteren Begrenzung und dem Vorzeichen von $v_y(t)$ ab. Dabei wird die y-Koordinate $s_y \bmod 60$ genau dann getroffen, wenn die Vertikalgeschwindigkeit des Balles $v_y(t)$ nicht negativ und die Abprallanzahl a geradzahlig, oder wenn die Vertikalgeschwindigkeit $v_y(t)$ negativ und Abprallanzahl a ungeradzahlig ist.

Analog kann auch die Aufschlagposition des Balles auf der gegnerischen Seite berechnet werden. Das Vorzeichen der Vertikalgeschwindigkeit lässt sich ebenfalls aus den Positionen des Balles zu 2 Zeitpunkten berechnen (sofern der Ball zwischen diesen Zeitpunkten nicht von einer Begrenzung abprallte). Außerdem lässt sich die Abprallanzahl a berechnen durch $a = \lfloor \frac{s_y - 30}{60} \rfloor$.

Schläger positionieren

Hat man einen Ball nun erfolgreich zurückgespielt, so wäre es beispielsweise denkbar, den eigenen Schläger wieder in die Ausgangsposition (Position, die der Schläger zum Zeitpunkt 0 inne hatte) zu bringen, um zu der nächsten Aufschlagposition schneller gelangen zu können. Eine noch effektivere Strategie kann es sein, den möglichen Bereich zu berechnen, in der die nächste Aufschlagposition liegen kann, und dann den Schläger in die Mitte dieses Bereiches zu steuern. Sieht man alle möglichen Abprallwinkel $\beta, \beta \in [-70°, +70°]$ vom gegnerischen Schläger jedoch als gleichwahrscheinlich an, so kann diese Mitte auch direkt berechnet werden, indem von einem Abprallwinkel von $\beta = 0°$ ausgegangen wird. Dies ergibt dann aber genau die Aufschlagposition auf den gegnerischen Schläger. Folglich kann der eigene Schläger einfach auch zu dieser bewegt werden.

Gewinnen?

Die bis jetzt vorgestellten Strategieelemente waren lediglich *defensiv*. Um ein Spiel jedoch auch aktiv gewinnen zu können, werden *offensive* Strategieelemente benötigt. Es muss also entschieden werden, in welchem Bereich des Schlägers der Ball abprallen soll. Diese Auswahl kann natürlich dem Zufall überlassen werden, aber auch die folgende Strategie ist denkbar. Dabei sei im Folgenden von einem optimal[10] spielenden Gegner ausgegangen. Spezielles Analysieren und Miteinbeziehen des gegnerischen Zugverhaltens ist an dieser Stelle auch denkbar, aber nicht notwendig.

Prinzipiell besitzt ein Spieler höchstens 3 Möglichkeiten, den Ball an seinem Schläger abprallen zu lassen: im oberen, mittleren oder unteren Bereich. Er hat also höchstens 3 Möglichkeiten für seinen *Metazug*. Offensichtlich vollziehen die Spieler ihre Metazüge immer abwechselnd. Es gilt also den Ball so zu steuern, dass der Gegner diesen *direkt* (nach dem nächsten eigenen Metazug) oder *indirekt* (also nach n Metazügen) nicht mehr mit seinem Schläger erreichen kann. Dabei ist der Ball für den Gegner direkt nicht mehr erreichbar, wenn die Zeit t_B, die der Ball bis zur seiner Aufschlagposition A_G auf der gegnerischen Seite benötigt, geringer ist als die Zeit t_S, die der gegnerische Schläger zu eben dieser Position A_G benötigt.

Um die Zeit t_B zu berechnen, kann die Bewegung des Balles als gleichförmig geradlinig aufgefasst werden. Da der Ball höchstens schneller wird, stellt die so erhaltene Zeit dann eine Abschätzung nach oben dar. Kann der Gegner jedoch schon in dieser Zeit seinen Schläger nicht zu der Aufschlagsposition A_G bewegen, so kann er dies erst recht nicht in einer geringeren Zeit. Sei $P'(64|p'_y)$ nun also der Schnittpunkt der Flugbahn des Balles $g: \binom{b_x}{b_y} + r \cdot \vec{v}(t), r \in \mathbb{R}$ mit der Parallen zur y-Achse durch 63, aus welchem sich bekanntermaßen die Aufschlagposition $P(-\hat{x}|p_y)$ berechnen lässt. Nach den bekannten Gesetzen der Physik ergibt sich für die Zeit t_B:

$$t_B = \frac{|\overline{P'B}|}{|\vec{v}|} = \frac{\sqrt{(b_x - 63)^2 + (b_y - p'_y)^2}}{\sqrt{v_x^2 + v_y^2}}$$

[10] Optimal im Sinne der nachfolgend vorgestellten Strategie.

Da sich Schläger höchstens geradlinig gleichförmig mit einer Geschwindigkeit $\vec{v_S}, |\vec{v_S}| = 1$ bewegen, ergibt sich für die Zeit t_S:

$$t_S = \max\left\{\frac{s_y - p_y}{1}, \frac{p_y - (s_y + 5)}{1}, 0\right\}$$

Eine Möglichkeit für einen eigenen Metazug heißt nun *gewonnen*, wenn der Gegner durch die Wahl dieser Möglichkeit den Ball mit seinem Schläger nun direkt nicht mehr erreichen kann. Analog heißt eine Möglichkeit für einen eigenen Metazug *verloren*, wenn der Gegner dann mit seinem nächsten Metazug eine Möglichkeit hat, die für ihn gewonnen ist, er den Ball also so steuern kann, dass der Spieler den Ball dann mit seinem Schläger nicht mehr erreichen kann.

Allgemein heißt ein eigener oder gegnerischer Metazug nun genau dann gewonnen, wenn dieser Zug eine gewonnene Zugmöglichkeit hat, oder es eine Zugmöglichkeit gibt, so dass der nächste Metazug des anderen Spielers dann nicht gewonnen ist. Ein eigener oder gegnerischer Metazug heißt verloren, wenn alle Zugmöglichkeiten dieses Metazuges verloren sind.

Im Spiel sollte nun also nach Möglichkeit ein gewonnene Möglichkeit gewählt werden. Bei der Berechnung des zu wählenden Zuges treten nun jedoch zwei Probleme auf. Zum einen lässt sich bei einem potenziell sehr lang andauernden Spiel nicht mehr berechnen, ob ein Zug gewonnen ist. Die zu erwartende Laufzeit von $\mathcal{O}(3^n)$ kann zwar vielleicht noch durch geschicktes Pruning teilweise verringert werden, wobei n die *Suchtiefe*, also die Anzahl an Metazügen bezeichnet, bis eine eigene gewonnene Möglichkeit auftritt. Generell ist es aber auch bei dem in konkreten Metazügen auftretenden Zufall wenig sinnvoll, diese Suchtiefe n nicht zu beschränken. Beschränkt man die Suchtiefe n jedoch nicht, so können zum anderen Metazüge auftreten, die weder gewonnen noch verloren sind. So kann es speziell auch dazu kommen, dass die Auswahl der als nächstes zu spielenden Zugmöglichkeit nicht mehr möglich ist, da keine Zugmöglichkeit gewonnen ist.

Um speziell dieses Problem zu lösen, kann jedem Metazug und jeder Zugmöglichkeit eine *Bewertung* zugeordnet werden. So erhielte z.B. ein gewonnener Zug die Bewertung ∞, ein verlorener eine von $-\infty$ und ein weder gewonnen, noch verlorener Zug in der Suchtiefe n eine Bewertung, welche umso größer ist, je wahrscheinlicher es für diesen Spieler ist, durch die Wahl dieses Zuges das Spiel zu gewinnen. So könnte diese Bewertung den Abstand der Aufschlagsposition des Balles bei dem Gegner zur Ausgangsposition berücksichtigen. Auch Berücksichtigungen der verschiedenen Wahrscheinlichkeiten der Abprallwinkel von Schlägern bei einer Zugmöglichkeit sind denkbar.

Ein Metazug erhält dann die Bewertung, die dem Maximum der Bewertungen der Zugmöglichkeiten entspricht, eine Zugmöglichkeit (sofern sie nicht schon nach den vorangegangenen Regeln bewertet wurde) das Negative der Bewertung des aus ihr resultierenden Metazuges. Letztendlich wird dann die Zugmöglichkeit mit maximaler Bewertung gezogen. Dieses Vorgehen wird auch als *MiniMax-Algorithmus* bezeichnet.

5.3 Bewertungskriterien

- Die in der KI realisierten Verfahren sollen nachvollziehbar beschrieben und ihre Wahl gut begründet sein.
- Die realisierte KI sollte mehr tun als einfach der Höhe des Balles zu folgen.
- Die von der KI verfolgte Strategie enthält neben defensiven auch zumindest Ansätze offensiver Elemente.
- Die Eigenschaften des Spiels wurden berücksichtigt, insbesondere die Beschleunigung des Balles und die unterschiedlichen Ausfallswinkel abhängig vom getroffenen Schlägerdrittel. Auch der Unterschied zwischen echter Position und Pixelposition des Balles sollte beachtet sein, aber ein diesbezüglicher Mangel reicht alleine nicht für einen Abzug.
- Eine funktionierende KI wurde im Turnierserver abgegeben. Die abgegebene KI sollte lediglich die bereitgestellten Methoden benutzen, um die Spielsituation auszulesen und den Zug abzugeben. Auch sollten generell keine eventuell gefundenen Fehler in der Turnierumgebung ausgenutzt worden sein. Allerdings sind diese Anforderungen schwer zu prüfen; gewertet wird im Zweifel für den Teilnehmer.
- Die abgegebene KI kann ihre Funktionalität im Turnier unter Beweis stellen und schneidet dort brauchbar ab. Wenn die KI überhaupt nicht abgegeben wurde (vgl. den vorigen Punkt), wird hier zusätzlich abgezogen.
- Spieldokumentation und Aufgabenstellung stehen bezüglich der Startgeschwindigkeit des Balles im Widerspruch: „Die Geschwindigkeit des Balls pro Schritt liegt zu Beginn bei 1.5 Pixel“ vs. „Zu Beginn bewegt sich der Ball mit einer Geschwindigkeit von einer Zellenseitenlänge pro Schritt“. Sollte dies zu Problemen bei z.B. der Umsetzung der Lösungsidee geführt haben, so wird dies nicht negativ bewertet.

Aus den Einsendungen: Perlen der Informatik

Allgemeines

Wort des Wettbewerbs: Allgoritmus

Man kann die Korrektheit des Programms beweisen, indem man auf die verschiedenen Rechtecke blickt.

Dies lässt sich mit kleinen Fallunterscheidungen ändern, das würde aber die Übersichtlichkeit, mit der das Programm ohnehin nicht gesegnet ist, nur schmälern.

Doch der schwierigste Teil unserer Aufgabe lag noch vor uns, wir hatten noch keine Ahnung, was uns in den kommenden Wochen erwartete.

So wird die Zählvariable i in Binär umgewandelt, dieser String wird in ein Char-Array umgewandelt, welches über den Umweg eines weiteren Character-Arrays in eine linked list umgewandelt wird.

Fähre füllen

Codeschnipsel: `min_laenge = 250 # schon ziemlich klein für ein Auto (Smart)`

Junioraufgabe 1: Faire Füllen *. . . , die fair die Fähre füllen – oder wie?*

Zahlenspiel

Hierzu haben wir leider keine Perlen gefunden.

Faires Füllen

Von irrationalen Werten wird nicht ausgegangen, da es sich bei der Aufgabenstellung um ein Rätsel handelt, welches Menschen gestellt wird. *. . . , und Menschen handeln immer rational . . .*

Es ist nun möglich, Aufgaben mit 3 Behältern oder mit 4 Behältern zu lösen, insofern es möglich ist.

Mobile

. . . , da die meisten echten Mobiles keine mehrere hundert Längeneinheiten lange Balken haben. *Kommt auf die Längeneinheit an . . .*

. . . , habe ich „aus Versehen“ ein Programm geschrieben, was die Ausgeglichenheit eines Mobiles überprüft.

. . . werden die ersten $n-1$ Abstände relativ willkürlich (ein bisschen Zufall ist schon dabei) ganzzahlig gesetzt.

Buffet-Lotterie

sillables *Echt albern, diese Silben!*

Beispiel-Abzählreim: Die In-tel-li-genz verfolgt dich, doch du bist schnel-ler.

Sobald das Geburtstagskind mindestens so oft an der Reihe war wie die Person, die essen gehen darf, Sitzpositionen in die Sprechrichtung entfernt ist, hätte das Geburtstagskind durch das Aussprechen zusätzlicher Silben essen gehen können.

Ausgabe für 23 Teilnehmer, 0 Silben: Hungerstreik? So kommt doch niemand zum Buffet!

Alphametiken

Alphemtiken

```
BIBER + GOES = BWINF
80817 + 4219 = 85036
```

```
BIBER + GOES = BWINF
16147 + 2543 = 18690
```

Da der Computer nicht denken kann, welche Kombinationen sinnvoll erscheinen, ist es notwendig, sämtliche Möglichkeiten auszuprobieren und zu überprüfen, ob eine davon zu einer Lösung kommt.

Mit dieser Formulierung verschleiern wir, dass es sich um einen Brute-Force-Algorithmus handelt.

Laut Duden können Zahlwörter bis 999.999 gebildet werden.

Für `SUCHEN - MACHT = SPASS` hat mein Programm als einzige Lösung [...] gefunden (ob es dabei Spaß hatte, hat es mir nicht mitgeteilt).

Es ist ein NP-Problem, denn die Lösung lässt sich leicht überprüfen, doch egal, wie sehr man die Arithmetik beachtet, man muss trotz dessen viele Kombinationen durchprobieren, es bleibt ein schweres Problem und wird nicht polynomiell schwerer, die benötigte Zeit lässt sich also nicht mit Hilfe eines Polynoms vorab estimieren. *Liest sich so, als hätte Heinrich von Kleist bei der Erstellung der Doku geholfen.*

Pong

Damit eine KI gewinnt, darf sie nicht verlieren.