27. Bundeswettbewerb Informatik, 1. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

Zuerst soll an dieser Stelle gesagt sein, dass wir uns sehr darüber gefreut haben, dass einmal mehr so viele Leute 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 miss achte. Im Einzelnen:

- Online-Anmelder wurden gebeten, ihre Nummer außen auf den Umschlag zu schreiben. Viele haben das gemacht, aber beinahe ebenso viele leider nicht. Das führt zu Komplikationen und möglicherweise zum Ausbleiben der versprochenen Rückmeldung per E-mail über den Eingang der Einsendung.
- Eine Gruppeneinsendung schicken Sie bitte komplett in einem gemeinsamen Umschlag, wir haben sonst größte Mühe, die Einsendungen richtig zuzuordnen. Wenn mehrere Einsendungen in einen Umschlag gesteckt werden, ist es besonders wichtig, bei der Online-Anmeldung bzw. auf den Anmeldebögen die Zusammensetzung der Gruppe anzugeben. Außerdem: Eine Gruppe muss sich auf eine Lösung pro Aufgabe einigen, und Gruppenmitglieder können nicht gleichzeitig auch eine eigene Einsendung schicken.
- Seien Sie mit Ihrem Namen nicht so geizig! Schreiben Sie ihn ruhig häufiger, z. B. auf das erste Blatt jeder Aufgabe und auf Ihre CD.
- Lösungsidee, Programm-Dokumentation und insbesondere auch Programm-Ablaufprotokolle und ausreichend viele Beispiele müssen ausgedruckt sein. Wir können aus Zeit- und Kostengründen keine Ausdrucke machen und auch nicht jedes eingesandte Programm ausführen.
- Noch schlechter als Einsendungen nur auf Datenträgern wären für uns übrigens Einsendungen via E-Mail oder anderen Internet-Wegen, auch wenn das für die Teilnehmer noch so praktisch wäre. Papiereinsendungen sind (zumindest zur Zeit und sicher auch noch in den nächsten Jahren) einfach unumgänglich.

- Beispiele werden als Teile des Programm-Ablaufprotokolls immer erwartet. Zu wenige Beispiele und erst recht die Nichtbearbeitung vorgegebener Beispiele führen zu Punktabzug. Es ist auch nicht ausreichend, Beispiele nur auf CD abzugeben, ins Programm einzubauen oder den Bewertern das Erfinden und Testen von Beispielen zu überlassen. Ohne abgedruckte Beispiele ist die Bewertung einer Lösung in der knappen vorhandenen Zeit nicht möglich. Leider fehlten in vielen Einsendungen die Beispiele, was oft das Erreichen der zweiten Runde verhindert hat.
- Zu einer Einsendung gehören auch lauffähige Programme. Kompilierung von Quellcode ist während der Bewertung nicht möglich. Für die gängigsten Skript-Sprachen stehen Interpreter zur Verfügung. Nutzen Sie aber bitte dennoch jede Möglichkeit, eigenständig ausführbare Programme abzugeben.

So, vielleicht denken Sie ja an diese Anmerkungen, wenn Sie (hoffentlich) im nächsten Jahr wieder mitmachen.

Auch die folgenden eher inhaltlichen Dinge sollten Sie beachten:

- Lösungsideen sollten Lösungsideen sein und keine Bedienungsanleitungen oder Wiederholungen der Aufgabenstellung. Es soll beschrieben werden, welches Problem hinter der Aufgabe steckt und wie dieses Problem grundsätzlich angegangen wird. Eine einfache Mindestbedingung: Bezeichner von Programmelementen wie Variablen, Prozeduren etc. dürfen nicht verwendet werden – eine Lösungsidee ist nämlich unabhängig von solchen Realisierungsdetails.
- Auch ein Programmablauf-Protokoll soll keine Bedienungsanleitung sein. Es beschreibt nicht, wie das Programm ablaufen sollte, auch nicht die zum Ablauf nötigen Interaktionen mit dem Programm, sondern protokolliert den tatsächlichen, inneren Ablauf eines Programms. Am besten protokolliert ein Programm seinen Ablauf selbst, z. B. durch Herausschreiben von Eingaben, Zwischenschritten oder -resultaten und Ausgaben.
- Oben wurde schon gesagt, dass Beispiele immer dabei sein sollten, zumindest eines davon in einem Programm-Ablaufprotokoll. Das hat seinen Grund: An den Beispielen ist oft direkt zu sehen, ob bestimmte Punkte korrekt beachtet wurden. Viele meinen nun, wir könnten die Programme ja laufen lassen und selbst auf Beispieldaten ansetzen, und liefern keine Beispiele oder nur Beispieldaten in elektronischer Form. Das können wir aber aus Zeitmangel in der Regel nicht. Außerdem ist nicht immer sicher, dass Programme, die auf dem eigenen PC laufen, auch auf einem anderen Computer ausführbar sind. Generell muss man sich darauf einstellen, dass nur das Papiermaterial angesehen wird!
- Mit den verschiedenen Beispielen sollten Sie wichtige Varianten des Programmablaufs zeigen, also auch Sonderfälle, die Ihre 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, bei Mängeln gibt es entsprechend weniger Punkte. Für die Gesamtbewertung sind die drei am besten bewerteten Aufgabenlösungen maßgeblich, es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreichen Sie mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten

und eine Anerkennung mit 9 bis 11 Punkten. Die Preisträger sind für die zweite Runde qualifiziert.

- Auf den Bewertungsbögen bedeutet ein Kreuz in einer Zeile, dass die (negative) Aussage in dieser Zeile auf Ihre Einsendung zutrifft. Damit verbunden ist dann in der Regel der Abzug eines oder mehrerer Punkte. Eine Wellenlinie bedeutet „na ja, hätte besser sein können“, führt aber meist nicht zu Punktabzug. Mehrere Wellenlinien können sich aber zu einem Punktabzug addieren. Gelegentlich sind lobende Anmerkungen der Bewerter mit einem '+' versehen.
- Wellenlinien wurden übrigens häufig für die Dokumentation (also Lösungsidee, Programm-Dokumentation, Programm-Ablaufprotokoll und kommentierter Programm-Text) verteilt, obwohl Punktabzug auch gerechtfertigt gewesen wäre.
- Aber auch so ließ sich nicht verhindern, dass etliche Teilnehmer nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden. Das ist ziemlich riskant, da Fehler sich leicht einschleichen.

Zum Schluss:

- Sollte Ihr Name auf der Urkunde falsch geschrieben sein, können Sie gerne eine neue anfordern. Uns passieren durchaus schon mal Tippfehler, und gelegentlich scheitern wir bei Anmeldungen auf Papierformular an der ein oder anderen Handschrift.
- Es ist verständlich, wenn jemand, der nicht weitergekommen ist, über eine Reklamation nachdenkt. Gehen Sie aber bitte davon aus, dass wir kritische Fälle, insbesondere die mit 11 Punkten, schon genau und mit Wohlwollen geprüft haben.

Danksagung

An der Erstellung der Lösungsideen haben mitgewirkt: Thomas Leineweber (Junioraufgabe), Benito van der Zander (Aufgaben 1 und 3), Thomas Bünger, Lilli Kaufhold und Marvin Künnemann (Aufgabe 2), Melanie Schmidt (Aufgabe 4) und Johann Felix von Soden-Fraunhofen (Aufgabe 5).

Die Aufgaben wurden vom Aufgabenausschuss des BWINF entwickelt, aus Vorschlägen von Hans-Werner Hein (Junioraufgabe), Monika Seiffert und Arno Pasternak (Aufgabe 1), Kirsten Riechmann (Aufgaben 2 und 5), Torben Hagerup (Aufgabe 3) und Melanie Schmidt (Aufgabe 4).

Junioraufgabe: Passendes Wort

Kriterien für gute Passwörter

Es gibt viele Kriterien, die gute Passwörter auszeichnen. Dabei kommt es insbesondere darauf an, dass das Passwort schwer zu erraten bzw. schwer zu bestimmen ist.

Länge: Je länger ein Passwort ist bzw. sein darf, desto größer wird der Suchraum für die Passwörter. Die Güte kann hier einfach durch Abzählen gemessen werden.

Zeichensatz: Wenn für ein Passwort mehr Zeichen erlaubt sind, wird dadurch der Suchraum pro Zeichen größer. Als Beispiele dafür gibt es Groß-/Kleinschreibung, Zahlen oder Sonderzeichen. Die Güte kann durch die Anzahl der verschiedenen Zeichen gemessen werden. Wenn aber z.B. verlangt wird, dass mindestens ein Zeichen eine Zahl sein muss oder mindestens ein Großbuchstabe vorhanden sein muss, dann ist dies im Prinzip wieder eine Einschränkung des Suchraumes, da für die entsprechende Position dann nicht mehr alle erlaubten Zeichen möglich sind.

Wörterbuch: Das Passwort sollte nicht einem bekannten Wort oder einer Abwandlung davon entsprechen (z.B. „F3uer“, verkürzt, verlängert, etc.). Diese können mit Hilfe einer sogenannten Wörterbuchattacke, die auch Varianten in den Wörtern ausprobiert, schnell entdeckt werden. Dies soll entsprechend auch für Teilpasswörter gelten. Ein Test, ob dieses Kriterium erfüllt ist, ist ohne ein entsprechendes Wörterbuch schwer zu implementieren. Dies betrifft auch die Bestimmung der Güte bzgl. dieses Kriteriums.

Merkbarkeit: Ein Passwort sollte für die Besitzerin/den Besitzer merkbar sein, also möglichst gut im Gedächtnis bleiben. Dieses Kriterium überschneidet sich mit dem Längenkriterium; sehr lange Passwörter kann man sich schlecht merken.

Ergonomie beim Schreibfluss/Schlechte Beobachtbarkeit: Das Passwort sollte schnell und gut eintippbar sein, damit ein Beobachter bei der Eingabe nicht einfach das Passwort mitlesen kann. Durch eine schnelle Eingabe wird das Mitlesen erschwert. Die Ergonomie eines Passwortes beeinflusst auch das obige Kriterium der Merkbarkeit; ein Passwort, das „gut in den Fingern liegt“, kann auch besser gemerkt werden.

Passwortgüte

Für die Berechnung der Passwortgüte müssen (nach Aufgabenstellung) zuerst drei Kriterien aus den vorher beschriebenen ausgewählt werden. Danach muss mit Hilfe dieser Kriterien die Gesamtgüte bestimmt werden. Dabei ist es möglich, zuerst einzelne Gütewerte zu den Kriterien zu bestimmen und diese dann zu einer Gesamtgüte zu verknüpfen. Dies wird aber nicht zwangsweise verlangt, es ist ebenso möglich, direkt eine Gesamtgüte zu bestimmen, ohne über die Einzelwerte zu gehen.

Wenn man mit positiven Einzelwerten arbeitet (je höher, desto besser), kann man z.B. durch eine Multiplikation eine Gesamtgüte bestimmen. Durch die Multiplikation wird vermieden,

Passwort	Wert 1	Wert 2	Wert 3	Gesamtgüte
aaA	3	2	2	6
aaaaaa	6	1	1	6
ababab	6	2	1	12
abcdefgh	8	8	1	64
ako2Hoo1	8	6	3	144
Evan#g5␣jo	10	10	5	500

Tabelle 1: Verknüpfung mit Multiplikation

Passwort	Wert 1	Wert 2	Wert 3	Gesamtgüte
aaA	3	2	2	7
aaaaaa	6	1	1	8
ababab	6	2	1	9
abcdefgh	8	8	1	17
ako2Hoo1	8	6	3	17
Evan#g5␣jo	10	10	5	25

Tabelle 2: Verknüpfung mit Addition

dass es eine hohe Gesamtgüte gibt, wenn ein Einzelwert besonders schlecht ist. Dies soll verhindern, dass man einzelne Kriterien bei der Passwortbestimmung vernachlässigt und dadurch wieder ein schlechtes Passwort bekommt.

Für die Implementierung reicht es vollkommen, mit den einfacheren Kriterien zu arbeiten. Eine einfache Kombination besteht aus der Passwortlänge, den verschiedenen vorkommenden Zeichenarten und den verschiedenen Zeichen überhaupt. Die Berechnung der Gesamtgüte muss dabei in der Implementierung die vorher beschriebene Gesamtgüte erkennen lassen.

Beispiel

Es werden die drei Kriterien „Passwortlänge“, „Anzahl verschiedener Zeichen“ und „vorkommende Zeichenarten“ ausgewählt. Für die ersten beiden Kriterien wird die Länge des Passwortes und die Anzahl der verschiedenen Zeichen als Wert genommen. Beim dritten Kriterium wird für jedes Zeichen des Passwortes geprüft, ob es eine Ziffer, ein Kleinbuchstabe, ein Großbuchstabe, Whitespace (Leerzeichen, etc.) oder ein anderes Zeichen ist. Für jede vorhandene Sorte von Zeichen gibt es einen Punkt. Die Summe der Punkte ist der Wert für das dritte Kriterium.

Wenn als Verknüpfung der Einzelwerte die Multiplikation der Einzelwerte gewählt wird, ergeben sich die in den Tabellen 1 bzw. 2 dargestellten Ergebnisse für beispielhafte Passwörter.

Insbesondere die Addition zeigt hierbei den Nachteil, dass niedrige Einzelwerte recht einfach durch hohe Werte eines anderen Kriteriums ausgeglichen werden können, so dass zwei Passwörter, von denen eines im Normalfall als deutlich sicherer angesehen werden wird (`abcdefgh` und `ako2Hoo1`) die gleiche Güte haben. Bei der Mulitplikation wirkt sich ein schlechter

Einzelwert dagegen so aus, dass es schwieriger wird, dies mit verbesserten anderen Werten auszugleichen.

Bewertungskriterien

- Bei Teilaufgabe 1 werden keine besonderen Anforderungen an die Begründung der Kriterien gestellt, sie müssen nur begründet sein.
- Es sollten mindestens fünf Kriterien genannt werden, wobei für jedes Kriterium eine Begründung und eine Einschätzung zur Implementierungsschwierigkeit geben muss. Einfachste Begründungen und Einschätzungen werden akzeptiert, sie dürfen nur nicht eindeutig falsch sein.
- Bei Teilaufgabe 2 können drei beliebige, auch sehr einfach implementierbare Kriterien genutzt werden.
- Sowohl die Kriterienauswahl als auch die Berechnung der Gesamtgüte muss (kurz!) beschrieben und begründet werden.
- Die Implementierung sollte auf Basis der Teilaufgabe 2 erfolgen.
- Die Implementierung sollte an mehreren Beispielausgaben die Gesamtgüte bestimmen. Die Ausgaben sollten mit einfachen Aussagen zu den Gütewerten versehen werden (Vergleiche, gutes Passwort, schlechtes Passwort, etc.).

Aufgabe 1: Pizzavision

Diese Aufgabe kann dazu verführen, sich im wesentlichen mit der grafischen Darstellung einer Pizza und ihrer Zutaten zu befassen. Diese spielt schon eine Rolle, aber der Kern der Aufgabe zielt auf die Benutzerinteraktion und das geometrische Problem, verschiedene Formen ausgewogen auf einem annähernd kreisförmigen Untergrund zu verteilen.

Lösungsidee

Benutzerinteraktion

Der Benutzer muss in der Lage sein (vielleicht ausgehend von einer Standardbelegung), Zutaten hinzuzufügen oder zu entfernen. Nach jedem Schritt muss das angezeigte Bild aktualisiert werden.

Ermöglicht es das Programm, die Zutaten frei auf der Pizza zu verschieben, muss für eine neue Zutat trotzdem automatisch eine sinnvolle Position auf der Pizza gewählt werden. Es reicht nicht aus, die gesamte Anordnung dem Benutzer zu überlassen oder alle Zutaten immer an derselben Stelle hinzuzufügen, so dass nur die letzten ohne manuellen Eingriff erkennbar wären.

Belegung der Pizza

Bei der Berechnung einer Zusammenstellung sind folgende Kriterien zu beachten: Die Zutaten sollten sich nicht auf einen Teil der Pizza beschränken, sondern einigermaßen ausgewogen auf ihr verteilt sein. Die Anwendung eines starren Rasters – insbesondere eines, das die (in der Regel) runde Form der Pizza ignoriert – führt allerdings zu allzu gleichmäßigen und damit eher unappetitlich wirkenden Belegungen.

Die einzelnen Bilder sollten sich nicht so großflächig überlappen, dass sie unerkennbar werden. Andererseits können sich die Zutaten an einigen Stellen schon berühren/überlappen, da dadurch die Bildkomposition interessanter wird. Man kann sich aufwändige Methoden überlegen, die einzelne Salamischeiben, Peperoni, Oliven, Tunfischkleckse etc. geschickt und je nach gegebener Belagkonstellation immer anders auf der Pizza verteilen. Es ist aber auch akzeptabel, wenn es ein festes Bild für jede Zutat gibt und auch eine feste Ordnung unter den Zutaten, die vorgibt, in welcher Reihenfolge die Bilder der aktuellen Zutatenkombination auf der Pizza platziert werden. Hier ist es dann praktisch „Handarbeit“, Ordnung und Bilder so aufeinander abzustimmen, dass die Pizzabilder möglichst gut aussehen.

Auch die Größenverhältnisse der Zutaten relativ zueinander und zum Pizzaboden sollten bedacht werden: sind die Bilder zu groß, belegt der Kunde die Pizza mit zu wenig Zutaten und ist später enttäuscht. Sind sie dagegen zu klein, wird die Pizza überfüllt (was im Prinzip allerdings für die Pizzeria besser ist).

Dann sollten keine grafischen Artefakte vorhanden sein wie beispielsweise einfarbige Hintergrundrahmen um alle Zutaten. Diese Bedingung lässt sich relativ einfach mittels transparenter

Layer erfüllen, man kann aber auch den Hintergrund der Zutatenbilder an den Pizzaboden anpassen. Bei Überlappungen müsste man aber ohne Transparenz Bilder für mehrere Kombinationen bereit haben.

Außerdem sollte die Pizza physikalisch möglich sein. Bei einem 2D-Bild müssen sich für diese Bedingung zumindest alle Zutaten auf der Pizza befinden, da man sie sonst nicht backen kann. Es ist auch gut, wenn das Problem gelöst wird, indem der Pizzaboden beim Hinzufügen neuer Zutaten von alleine wächst. Bei einer 3D-Darstellung, mit den Zutaten als einzelne 3D-Modelle, sollte sichergestellt werden, dass keine Zutaten in der Luft schweben, ineinander verschoben sind oder einen Turm bilden. (Mit einem solchen Modell könnte man auch den Backvorgang in physikalischer Hinsicht als Funktionen auf einem Partikelsystem darstellen und so die jeweilige Knusprigkeit [z.B.: das Wasser aus Tomaten wirkt der Knusprigkeit entgegen] oder die optimale Backzeit berechnen. Interessant ...)

Letztendlich muss es auf jeden Fall Käse unter den Zutaten geben, da eine Pizza ohne Käse nicht schmeckt. ;-)

Ablaufdokumentation

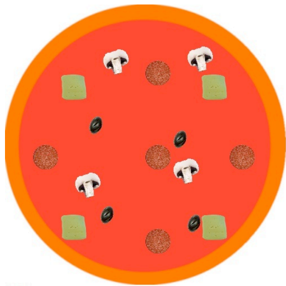
Bei der Dokumentation des Ablaufs sollten Bilder der Pizza gezeigt werden, bei denen klar erkennbar wird, dass aus mindestens zwölf Zutaten gewählt werden kann. Außerdem sollte kurz beschrieben werden, wie die gezeigte Anordnung entstanden ist, also ob sie automatisch generiert ist oder manuell nachbearbeitet wurde und in welcher Reihenfolge die Zutaten hinzugefügt wurden. Auch das geforderte Entfernen einer Zutat muss gezeigt werden.

Bewertungskriterien

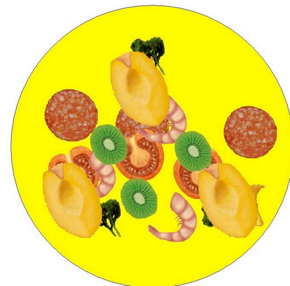
- Eine automatische Platzierung der Zutaten muss möglich sein.
- Der Benutzer muss aus mindestens zwölf Zutaten auswählen können.
- Das Bild muss nach jeder Veränderung der Zutatenliste (Hinzufügen oder Entfernen) aktualisiert werden.
- Die Zutaten müssen einigermaßen ausgewogen platziert werden, also in gleichmäßiger Verteilung, ohne zu große Überlappungen und ohne Anwendung eines allzu starren Rasters (Raster-Pizzen sehen einfach nicht lecker aus).
- Die Platzierung ergibt eine „backbare“ Pizza (z.B. alle Zutaten auf der Pizza).
- Die dokumentierte Zutatenauswahl sollte mindestens drei Zutaten (außer Käse) beinhalten und auch den Effekt des Entfernens einer Zutat demonstrieren.

Pizzabeispiele

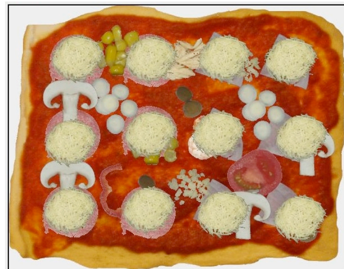
Hier einige Beispiele für Pizzaabildungen.



Anordnung der Zutaten auf sehr gleichmäßigem Raster.



Auf dieser Pizza überlappen sich die Zutaten stark.



Eine rechteckige Pizza macht die Sache einfacher. Ist aber OK, wenn es gut aussieht.

Einfach, aber durchaus den Ansprüchen genügend.

Und diese Pizza ist richtig gut gelungen – lecker!

Aufgabe 2: Tankomatik

Lösungsidee

Teilaufgabe 1: Simulation der Preisentwicklung

Der erste Schritt zur Lösung dieser Aufgabe ist es, sich Gedanken über „realistische“ Preisentwicklungen zu machen. Vor allem folgende zwei Kriterien dürften nahe liegen:

- Es treten in der Regel keine zu großen Preissprünge auf.
- Die generelle Preistendenz ist steigend.

Optimisten dürfen natürlich auch eine schwankende Preistendenz einbauen – derzeit entspricht sogar eine fallende Tendenz der Realität.

Die Realisierung einer solcher Preisentwicklung könnte so aussehen: Mit einer frei wählbaren Wahrscheinlichkeit von $p \geq \frac{1}{2}$ steigt der Preis, ansonsten sinkt er oder bleibt gleich. Danach folgt die Bestimmung der Höhe der Differenz des Preises. Eine Favorisierung der niedrigen Beträge wäre hier wünschenswert. Folgendes Verfahren wäre dafür passend: Man wählt eine Wahrscheinlichkeit q. Ausgehend von einem Startbetrag von 0 ct wird so lange eine zufällige reelle Zahl zwischen 0 und 1 erzeugt, bis die Zahl kleiner als q ist. Jedes Mal, wenn sie größer als q ist, wird der Betrag um 1 ct erhöht. Der resultierende Betrag wird die neue Preisänderung.

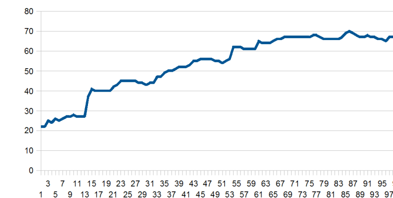


Abbildung 1: Die Abbildung zeigt ein Beispiel einer Preisentwicklung über 100 Tage.

Auch andere sinnvolle Kriterien sind vorstellbar. Wichtig ist, dass diese besprochen und in der Diskussion der zweiten Teilaufgabe genutzt werden. Eine einfache, aber schlechte Möglichkeit ist die Wahl von beliebigen Zufallswerten innerhalb eines über den ganzen Simulationszeitraum hinweg gleich bleibenden Intervalls. Die Preisentwicklung fällt so zu sprunghaft aus.

Teilaufgabe 1: Simulation des Tankverhaltens

Für den Takt der Simulation sind Sekundenschritte sinnvoll, da größere Abweichungen von den durchschnittlich 60 Autos pro Stunde die Preisentwicklung bei Asso beeinflussen. Der größte akzeptable Takt ist eine Stunde, da immer zur vollen Stunde die Verkaufspreise festgelegt werden. Bei einem Stundentakt ist es denkbar, die Tankwahrscheinlichkeit zu ignorieren und die Anzahl der tankenden Autos auf 60 festzulegen. Dann kann mit je 12 Stammkunden und 36 preisorientierten Kunden gerechnet werden. Diese Vereinfachung ist zwar nicht ganz im Sinn der Aufgabenstellung, aber dennoch akzeptabel, da sie die Simulationsergebnisse nur geringfügig beeinflusst.

Spielraum besteht noch in der Festlegung des Verhaltens der Simulation, wenn Asso und Scholl zu gleichen Preisen verkaufen und wie der Startpreis Assos festgelegt wird. Die wohl natürlichste Annahme ist, dass sich die Kunden bei Preisgleichheit gleich auf beide Tankstellen verteilen. Weiterhin wäre es möglich, die Kunden zur günstigeren Tankstelle der letzten Stunde gehen zu lassen (Simulation dessen, dass die Kunden sich für die in der Regel günstigere Tankstelle entschieden haben). In diesem Fall müsste man den Effekt der Wahl besonders diskutieren.

Die Wahl des Startpreises Assos dürfte auf lange Sicht keine zu großen Auswirkungen haben, solange er nicht zu weit vom Einkaufspreis entfernt ist. Nahe liegend sind hier Werte wie der Einkaufspreis selbst.

Teilaufgabe 2: Wer macht mehr Gewinn?

Hier ist es gefragt, die Simulation zu nutzen und zu ermitteln, wie sich die Preisentwicklung auf die Umsätze und entsprechend die Gewinne der Tankstellen auswirkt. Im Zusammenhang mit der Berechnung des Gewinns tun sich noch weitere offene Punkte auf, die Festlegungen erfordern. Der Gewinn hängt insbesondere von den Ausgaben der Tankstellen beim Nachkauf von Benzin ab. Wann Benzin nachgekauft werden muss, wird insbesondere von der Kapazität der Tankstelle und auch von den Tankmengen der Kunden beeinflusst. Einfache Regeln sind hier akzeptabel: etwa eine feste Tankmenge (z.B. 30 Liter) und ein tägliches Auffüllen der Tankstellenvorräte – es wird also genau so viel nachgekauft wie getankt wurde.

Insgesamt kann man bei der Gewinnberechnung folgendes feststellen:

- Steigt der Preis sehr langsam an und macht keine Sprünge, ist Asso leicht im Vorteil, weil deren Preis nach dem Abrufen des Einkaufspreises knapp unter dem Scholls liegt und sich somit die meisten Kunden für Asso entscheiden. Da der Preis aber nur knapp drunter liegt, ist der Verdienst dabei noch relativ gut. Während der sechs Stunden steigt Assos Preis so lange, bis er den von Scholl erreicht hat. Dann bekommt jede Tankstelle etwa die Hälfte der Kunden, also 30, und der Preis bleibt stabil.
- Ansonsten ist Scholl im Vorteil, weil sie sich viel schneller an einen neuen Einkaufspreis anpassen. Sinkt dieser, ist der Preis von Asso meist über dem von Scholl und sie bekommen weniger Kunden. Steigt der Preis stark, setzt Asso seinen Preis auf den Einkaufspreis und macht somit erst wieder Gewinn, wenn nach einer Stunde der Preis um 1 ct angehoben wird.

Auf unser Modell der realistischen Preisentwicklungen angewandt macht Asso also mehr Gewinn als Scholl, da die Preise generell leicht ansteigen und große Preisanstiege und vor allem Preisstürze eher selten sind. Bei Preisentwicklungen, die um einen Wert schwanken, macht Scholl meist mehr Gewinn. Die Antwort hängt also stark von der Preisentwicklung ab!

Teilaufgabe 3: Assos Strategie

Eine Möglichkeit ist es, die Grenzen anzupassen, nach denen Asso bestimmt, ob der Preis sinkt oder steigt. Der Verdienst kann beispielsweise gesteigert werden, wenn Asso den Preis schon bei 30 Kunden und nicht erst bei weniger als 25 senkt. Wenn beide Tankstellen gleich teuer sind, verteilen sich die Kunden nach unserem Modell zu gleichen Teilen, d.h. Asso hat pro Stunde durchschnittlich 30 Kunden, da insgesamt 60 zu erwarten sind. In dem Fall wäre es viel günstiger, mit dem Preis wieder einen Cent runter zu gehen und so einen Großteil der Kunden zu sich zu locken. Diese Strategie hat hauptsächlich dann eine Wirkung, wenn Asso und Scholl häufig den gleichen Preis verlangen, was aber nur der Fall ist, wenn der Einkaufspreis keine großen Sprünge macht. Dieses Problem lässt sich beseitigen, indem Asso zusätzlich dynamisch mit dem Einkaufspreis mitgeht, d.h. wenn der Einkaufspreis um beispielsweise 4 ct steigt, erhöht Asso auch um 4 ct.

Mit Hilfe dieser beiden Strategien macht Asso unabhängig von der Entwicklung des Einkaufspreises deutlich mehr Gewinn als Scholl. Im Rahmen des Modells ist auch noch eine weitere Strategie denkbar, mit der sich der Gewinn sogar beliebig steigern lässt:

Asso verlangt für einen Liter Benzin 1000000 ct oder noch mehr, schließlich werden 20% der Kunden trotzdem weiter dort tanken. Dies ist natürlich nur eine rein theoretische Strategie, da selbst die treuesten Stammkunden sich im wahren Leben irgendwann von ihrer Tankstelle abkehren. Darauf sollte auch in der Einsendung hingewiesen werden.

Bewertungskriterien

- Die Preissimulation ist Grundlage der Tanksimulation und sollte einen einigermaßen stetigen, aber mit leichten Schwankungen versehenen Preisverlauf ergeben.
- Festlegungen zur genauen Umsetzung des Modells, also insbesondere Entscheidungen über in der Aufgabenstellung offen gelassene Punkte, müssen beschrieben und begründet werden.
- Die Simulation muss korrekt nach den beschriebenen Festlegungen realisiert sein.
- Die Frage, wer langfristig mehr Gewinn macht, muss abhängig von der Art der gewählten Preisentwicklung diskutiert werden, und die Antwort muss mit Simulationsergebnissen belegt sein.
- Mindestens eine mögliche andere Preispolitik Assos muss vorgestellt und diskutiert werden. Dabei muss darauf geachtet werden, dass kein Wissen über Scholls Preispolitik genutzt wird. Es ist schön, wenn die Vorschläge zu diesem Punkt auch in der Simulation umgesetzt und Vorhersagen mit Simulationsergebnissen belegt werden. Zumindest sollten die Vorschläge aber auf Erkenntnissen aus Teilaufgabe 2 beruhen.

Aufgabe 3: Alle Alpen

Lösungsidee

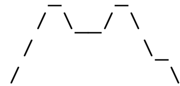
Teilaufgabe 1: Interne Darstellung Natürlich ist mit der „Darstellung in einer Programmiersprache“ nicht die grafische Darstellung mit Hilfe eines Programms gemeint, sondern die Repräsentation als Datenstruktur. Zur Darstellung des Gebirgszuges im Programm gibt es grundsätzlich zwei Möglichkeiten: Entweder man speichert für jeden Punkt die jeweilige Höhe, oder man speichert für jeden Punkt die Höhendifferenz zum vorherigen. Letzteres ist flexibler (man könnte Teile ausschneiden, etc.), was aber für die Aufgabe keine Rolle spielt. Um diese Daten im Speicher abzulegen, hat man die freie Wahl zwischen einem Array oder einer verketteten Liste. Speichert man den Gebirgszug aber als verkettete Liste, sollte man in den folgenden Aufgaben jeweils einen Zeiger auf das aktuelle Element verwenden und es nicht in jedem Schritt von Beginn an durchsuchen.

Teilaufgabe 2: Gebirgszüge zeichnen Bei der grafischen Ausgabe muss die Datenstruktur durchlaufen und für jeden Punkt die entsprechende Verbindung zum vorherigen gezeichnet werden. Die Ausgabe kann dabei auf dem Bildschirm oder in einer Datei (bmp, svg, ...) erfolgen, es sollte aber sichergestellt sein, dass man den ganzen Gebirgszug sehen kann. Auch bei größerer Länge (etwa $N = 100$) darf das Bild nicht abgeschnitten werden.

Auch ASCII-Art ist möglich, allerdings sollten zur besseren Anschaulichkeit mehrere Zeilen verwendet werden. Statt

```
///-\--/-\\-\
```

sollte also



ausgegeben werden.

Überflüssige Bildelemente wären zum Beispiel Bäume, Wolken, Bergsteiger, Hütten, Lawinen oder Skilifte. Es sollte aber nur dann ein Punkt abgezogen werden, wenn diese das Erkennen der Höhenstruktur stark erschweren.

Teilaufgabe 3

Gebirgszüge aufzählen Hier genügt eine einfache Backtrackinglösung, bei der alle möglichen Werte der Datenstruktur durchprobiert werden. Wenn 'pos' die aktuelle Position im Gebirgszug und 'gebirgszug' der aktuelle Gebirgszug ist, kann man den geforderten Algorithmus so beschreiben (der mit 'berechne(1)' augerufen werden müsste):

```
function berechne(pos)
  if (pos > n) and gueltig(gebirgszug) then P(gebirgszug)
  for h := moegliche Hoehenwerte
    do setze_hoehe(pos, h)
       call berechne(pos + 1)
end
```

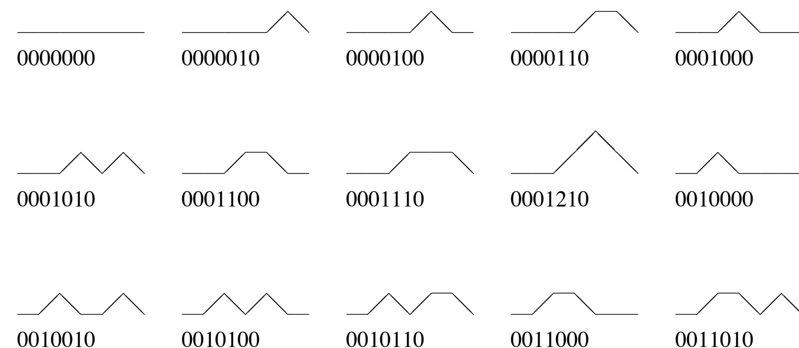

Speichert man die tatsächlichen Höhendaten, muss in jedem Schritt getestet werden, ob diese Belegung zu einem gültigen Gebirgszug fortgesetzt werden kann – insbesondere darf die Höhe an einer Position *pos* den Wert $N - pos$ nicht überschreiten. Wird dies nämlich erst getan, wenn ein Gebirgszug die Länge N erreicht, so beträgt die Laufzeit $O(N^N)$ anstatt $O(3^N)$, da es in jedem Schritt N statt 3 Möglichkeiten zur Auswahl einer Höhe gibt. Speichert man dagegen nur die Höhendifferenz, darf diese Prüfung auch erst am Ende erfolgen, da dies keinen großen Laufzeitunterschied ergibt.

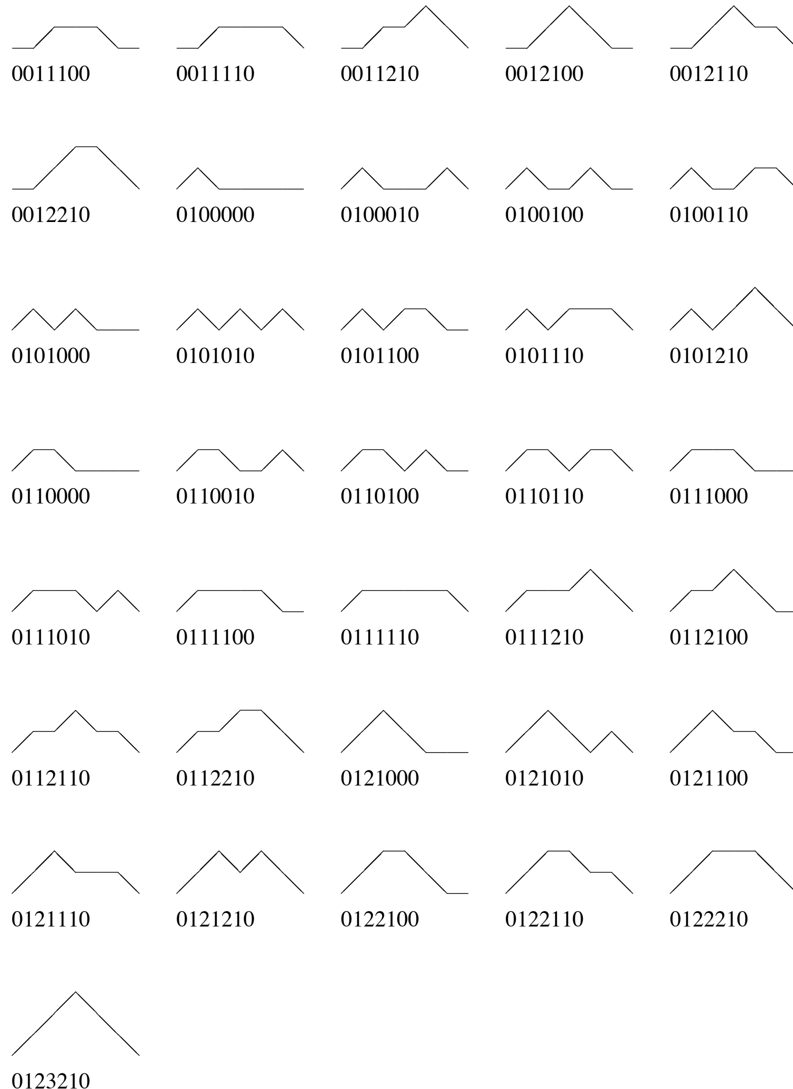
Die Prozedur P sollte idealerweise als Zeiger bzw. Referenz übergeben werden. Im Rahmen der ersten Runde ist es auch akzeptabel, wenn zunächst eine „leere“ Prozedur namens P deklariert wird. Aber es ist möglich, den aktuellen Gebirgszug als globale Variable zu übergeben oder komplett zu kopieren (Verlangsamung um $O(N)$).

Gebirgszüge der Länge N sind, wie in der Aufgabenstellung definiert, solche mit N Teilstücken, also $N + 1$ Höhenwerten. Gebirgszüge mit N Höhenwerten haben also die Länge $N - 1$. Wenn dies verwechselt wird, liefern ansonsten korrekte Aufzählungsverfahren in den folgenden Teilaufgaben Ausgaben für die Werte 5 (das sind 21 verschiedene Gebirgszüge) bzw. 15 (korrekter Wert: 310572). Wenn man von der vorgegebenen Definition abweicht und erklärt, dass Höhenwerte außer dem ersten und letzten nicht 0 sein dürfen, gibt es nur 9 bzw. 113634 verschiedene Gebirgszüge der Länge 6 bzw. 16 – was genau den regelgerechten Werten für 4 und 14, also für $N - 2$ entspricht, wie man sich leicht überlegen kann.

Gebirgszüge der Länge 6 ausgeben Zur Ausgabe kann der Gebirgszug entweder gleich angezeigt oder zwischengespeichert werden. Im ersten Fall können Sie alle untereinander angezeigt/abgespeichert werden oder jeder wird während der Berechnung einzeln für sich angezeigt und überschreibt die Ausgabe des vorherigen. Dann muss aber eine genügend lange Pause gegeben sein. Im zweiten Fall müssen die Gebirgszüge nach der Berechnung entsprechend angezeigt werden.

Es ist positiv, wenn die Methode aus 3.2 verwendet wird, aber nach der Aufgabenstellung nicht notwendig, also muss die Ausgabe auch nicht anschaulich sein. Lösungen (51):





Anzahl der Gebirgszüge der Länge 16 Die Aufgabenstellung verlangt, dass hierfür wieder alle möglichen Gebirgszüge erzeugt werden. Wer aber eine elegantere Lösung mittels

DP oder einer Formel wie

$$\left\lfloor \frac{1}{N+1} \sum_{k=0}^{\lceil (N+1)/2 \rceil} \binom{N+1}{k} \binom{N+1-k}{k-1} \right\rfloor$$

(von http://www.research.att.com/∼njas/sequences/A001006; die Gebirgszzug-Anzahlen abhängig von der Länge sind auch als Motzkin-Zahlen bekannt) findet, sollte trotzdem die Punkte erhalten.

Für die Prozedur P reicht es, einfach einen Zähler zu inkrementieren. Wichtig ist, dass nicht sämtliche Gebirgszüge während der Berechnung gespeichert werden, da dies bei größeren N zuviel Speicher verbraucht. Lösung: 853467

Bewertungskriterien

- Die programmiersprachliche Darstellung muss angemessen sein, darf also insbesondere die Effizienz der Lösung nicht unnötig behindern. Ein Beispiel (vgl. oben) ist, dass bei einer verketteten Liste ein Zeiger auf das aktuelle Element mitgeführt werden sollte. „Darstellung“ haben einige Teilnehmer als bildhafte Darstellung missverstanden. Dies ist ein Mangel, wenn deswegen keine Beschreibung der programmiersprachlichen Darstellung vorhanden ist. Es ist akzeptabel, wenn letztere in solchen Fällen erst in der Programm-Dokumentation zu finden ist.
- Die grafische Darstellung der Gebirgszüge muss übersichtlich sein und auch für die geforderte Länge ($N = 100$) funktionieren.
- Die Aufzählung der Gebirgszüge sollte nicht unnötig ineffizient sein. Eine Lösung mit purem „brute force“ (z.B. Zulassen unmöglicher Höhenwertfolgen mit erst nachträglicher Prüfung) ist nicht akzeptabel. Auch das erwähnte Zwischenspeichern von Gebirgszügen ist zumindest für größere N zu vermeiden.
- Die Aufzählung der Gebirgszüge und die in den Teilaufgaben 3a und 3b geforderten Prozeduren müssen korrekt implementiert sein.
- Die Ergebnisse für die Teilaufgaben 3a und 3b müssen korrekt sein. Liefert ein Programm die korrekten Werte für $N-1$ oder $N-2$, weil die Definition eines Gebirgszuges offensichtlich missverstanden oder angepasst wurde, ist das akzeptabel.
- Es genügt, wenn nur die geforderten Beispielfälle dokumentiert sind: Zeichnung eines Gebirgszuges der Länge 100, Ausgabe aller Gebirgszüge der Länge 6 („Ausgabe“ ungleich „Zeichnung“; eine Ausgabe der Zahlenfolgen ohne bildhafte Darstellung der Gebirgszüge ist akzeptabel!), Berechnung der Anzahl der Gebirgszüge der Länge 16. Die Gebirgszüge der Länge 6 sollten einigermaßen kompakt dokumentiert werden.

Aufgabe 4: Kreisrund

Problemstellung

Aufgabenstellung

Bei dieser Aufgabe geht es um Hugo Langbein, der von einigen seiner Kollegen Geld einsammeln muss. Dazu muss er sie während ihrer Teepausen abfangen. Jeder Mitarbeiter, den Hugo treffen muss, hat eine **Teezahl** t und hält sich alle t Minuten in einer Teeküche auf (in welche Teeküche ein Mitarbeiter geht, wissen wir aus der Eingabe). Alle Mitarbeiter beginnen gleichzeitig mit der Arbeit, und ein Mitarbeiter mit Teezahl t betritt seine Teeküche zum ersten Mal t Minuten nach Arbeitsbeginn. Hugo startet seine Einsammelaktion auch zum Arbeitsbeginn. Damit er einem Kollegen Geld abknüpfen kann, muss er entweder gleichzeitig mit diesem in einer Teeküche ankommen oder dort schon auf ihn warten.

Hugos Firma befindet sich im obersten Stock eines ringförmigen Gebäudes, und die Teeküchen befinden sich auf dem Ring gleichmäßig verteilt. Dabei sind die Teeküchen aufsteigend nummmeriert und neben Teeküche n liegt wieder Teeküche 1.

Unsere Aufgabe besteht darin, festzustellen, ob Hugo es innerhalb einer vorgegebenen Zeit schaffen kann, eine vorgegebene Liste von Mitarbeitern abzuklappern. Wir müssen insbesondere *nicht* die optimale Zeit berechnen, in der Hugo es schaffen kann, alle zu besuchen, sondern können aufhören, wenn wir eine Lösung gefunden haben, die innerhalb des gewünschten Zeithorizontes liegt (das klappt natürlich nur, wenn es eine solche Lösung gibt).

Grundlegende Überlegungen

Wir schauen uns jetzt eins der vorgegebenen Beispiele an, um die Aufgabenstellung zu veranschaulichen. Wir werden das Beispiel auch im nächsten Abschnitt noch benötigen. Zu Beispiel 2 war bei der Aufgabe auch die Lösung gegeben. Die Eingabedatei `beispiel2.in` enthält die folgenden Informationen:

```
7200 27
Annalena 57 2
Jan 59 15
Christian 58 22
```

Wir stellen zunächst einmal fest, dass die Eingaben in verschiedenen Einheiten sind. Das Zeitlimit 7200 ist in Sekunden, d.h. Hugo hat 120 Minuten bzw. 2 Stunden Zeit. Die Teezahlen hingegen sind in Minuten angegeben: Annalena trinkt z.B. alle 57 Minuten einen Tee. Dabei hält sie sich in Teeküche 2 auf. Insgesamt gibt es 27 Teeküchen.

Laufzeit zwischen zwei Teeküchen

Wie lange braucht Hugo jetzt, um z.B. von Teeküche 8 zu Teeküche 24 zu gelangen? Das hängt davon ab, ob Hugo den Ring von Teeküche 8 aus so abläuft, dass er zuerst bei 9 vorbeikommt, oder ob er andersherum läuft, also in Richtung der Teeküche 7. Im ersten Fall muss Hugo die Strecke $8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24$ zurücklegen, um zu Küche 24 zu gelangen, im anderen Fall die Strecke $8,7,6,5,4,3,2,1,27,26,25,24$. Da wir wissen wollen, ob Hugo es innerhalb der vorgegebenen Zeit schaffen *kann*, alle Mitarbeiter zu treffen, gehen wir davon aus, dass er immer den kürzeren Weg nimmt. Er würde also in Richtung der Teeküche 7 loslaufen. Dabei muss er elf mal von einer Teeküche zu einer anderen laufen, was 20 Sekunden dauert. Insgesamt benötigt er dann 220 Sekunden.

Wir möchten gern allgemein ausrechnen können, wie lange Hugo für die Strecke zwischen zwei Teeküchen benötigt. Sei dazu i die kleinere der beiden Teeküchennummern und j die größere. Es ist egal, ob Hugo von i nach j oder von j nach i laufen möchte, beides benötigt die gleiche Zeit. Deshalb stellen wir uns vor, Hugo läuft von i nach j. Wenn Hugo von i aus in Richtung $i+1$ losläuft, muss er die Strecke $i, i+1, i+2, \ldots, j-1, j$ zurücklegen. Man kann sich überlegen, dass er dabei genau $j-i$ mal an einer Teeküche vorbeikommt, d.h. er benötigt in diesem Fall $20*(j-1)$ Sekunden. Läuft Hugo andersherum, so legt er die Strecke $i, i-1, \ldots, 1, n, \ldots, j+1, j$ zurück. Dabei muss Hugo $i-1$ mal von einer Teeküche zur nächsten laufen, bis er bei Teeküche 1 ankommt. Läuft er eine Teeküche weiter, so kommt er zu Nummer n, und von da aus muss er noch $n-j$ mal von einer Teeküche zur nächsten laufen, bis er bei j ankommt. Insgesamt benötigt er also $20*(i+n-j)$ Sekunden, wenn er diesen Weg nimmt.

Um die kürzere Zeitspanne zu ermitteln, muss man das Minimum der beiden Zeiten nehmen. Wir können also festhalten, dass Hugo $20 \cdot \min\{(j-i),(i+n-j)\}$ Sekunden benötigt, um von Teeküche i nach Teeküche j zu gelangen (oder andersherum).

Dauer einer Route

Wir wissen jetzt, wie lange es dauert, von einer bestimmten Teeküche zu einer anderen Teeküche zu gelangen. Jetzt überlegen wir uns, wie lange es dauert, wenn Hugo sich vornimmt, die Mitarbeiter in einer bestimmten Reihenfolge zu besuchen. Dazu ziehen wir wieder unser Beispiel heran und nehmen an, Hugo hat sich überlegt, zuerst Christian zu treffen, anschließend Jan und zum Schluss Annalena.

Hugo läuft also zuerst von Teeküche 1 zu Teeküche 22 und braucht dafür $20 \cdot \min\{21,3\} = 60$ Sekunden. An dieser Stelle ist die Zeit, die Hugo zum Laufen benötigt, aber relativ unbedeutend: Christian erscheint ja erst nach 58 Minuten in der Teeküche. Hugo läuft also eine Minute, wartet 57 Minuten und erhält das Geld von Christian. Dafür benötigt er keine Zeit, also kann er 58 Minuten nach Arbeitsbeginn weiterlaufen und geht jetzt zu Teeküche 15, was $20 \cdot \min\{22-15, 15+24-22\} = 140$ Sekunden dauert. Und jetzt spielt die Zeit, die Hugo für den Weg benötigt hat, plötzlich eine sehr große Rolle: Jan kommt in die Teeküche genau 59 Minuten nach Arbeitsbeginn, und um ihn abzufangen, muss Hugo vor ihm da sein oder gleichzeitig mit ihm eintreffen. Hugo kommt aber erst 20 Sekunden später an, weil er 140 Sekunden braucht, um den Weg zurückzulegen. Er trifft Jan also nicht an und muss warten, bis dieser

118 Minuten nach Arbeitsbeginn das nächste mal in der Teeküche eintrifft. Zum Zeitpunkt 118 geht Hugo dann zu Teeküche 2 weiter, wofür er $20 \cdot \min\{15-2, 2+24-22\} = 80$ Sekunden benötigt. Dort trifft er dann zum Zeitpunkt 171 Annalena und hat das Geld eingesammelt – aber leider 51 Minuten zu spät.

Die ebenfalls veröffentlichte Beispieldatei verrät uns, dass es für das Beispiel tatsächlich keine Lösung gibt, die mit 120 Minuten auskommt.

Interner Umgang mit Zeiten

Um Verwirrung mit verschiedenen Einheiten zu vermeiden, sollte man programmintern alle vorkommenden Zeiten in eine Einheit umrechnen. Minuten eigenen sich dabei nicht so gut, weil 20 Sekunden $\frac{1}{3}$ Minute entsprechen und schlecht gespeichert werden können. Stattdessen kann man Sekunden als Grundeinheit verwenden oder alternativ auch direkt ausnutzen, dass Hugo für den Weg zwischen zwei beliebigen Teeküchen immer eine Zeit benötigt, die ein Vielfaches von 20 Sekunden ist. Man kann also statt Sekunden oder Minuten als Grundeinheit auch 20 Sekunden verwenden und erhält dadurch schöne glatte und kleine Zahlen.

Lösungsansätze

Backtracking (und Rekursion)

Wir möchten unser Problem mit **Backtracking** lösen. Dabei handelt es sich um ein bestimmtes Lösungsverfahren, das bei Wikipedia[1] folgendermaßen erklärt wird:

> Der Begriff Rücksetzverfahren oder englisch Backtracking [...] bezeichnet eine Problemlösungsmethode innerhalb der Algorithmik.
>
> Backtracking geht nach dem Versuch-und-Irrtum-Prinzip (trial and error) vor, d. h. es wird versucht, eine erreichte Teillösung schrittweise zu einer Gesamtlösung auszubauen. Wenn absehbar ist, dass eine Teillösung nicht zu einer endgültigen Lösung führen kann, wird der letzte Schritt bzw. die letzten Schritte zurückgenommen, und es werden stattdessen alternative Wege probiert. Auf diese Weise ist sichergestellt, dass alle in Frage kommenden Lösungswege ausprobiert werden können. Mit Backtracking-Algorithmen wird eine vorhandene Lösung entweder gefunden (unter Umständen nach sehr langer Laufzeit), oder es kann definitiv ausgesagt werden, dass keine Lösung existiert. Backtracking wird meistens am einfachsten rekursiv implementiert und ist ein prototypischer Anwendungsfall von Rekursion.

Wir sehen im nächsten Abschnitt konkrete Beispiele für Backtracking als Lösungen für unsere Aufgabenstellung.

Die Erklärung für Backtracking benutzt den Term **rekursiv**, den wir noch erläutern wollen, bevor wir zu unserer eigentlichen Problemstellung zurückkehren. Eine Prozedur oder Funktion heißt **rekursiv**, wenn sie sich selbst aufruft. Es ist oft kompliziert, rekursive Programme zu

[1] http://de.wikipedia.org/wiki/Backtracking

verstehen, obwohl das Grundprinzip einfach ist. Ein kurzes rekursives Programm könnte zum Beispiel so aussehen:

```
1 Prozedur rekursiveProzedur()
2     Gib den Text "Ein Aufruf!" auf dem Bildschirm aus aus
3     rekursiveProzedur()
```

Die Prozedur *rekursiveProzedur*() gibt einen Text aus und ruft sich anschließend selbst auf. Das bedeutet, dass das Programm niemals endet (wenn es nicht abgebrochen wird), sondern immer wieder die gleiche Prozedur startet und den Text ausgibt.

Sinnvolle rekursive Programme sollten irgendwann stoppen. Dazu benötigt man eine **Abbruchbedingung**. Auch hierfür sehen wir uns ein kleines Beispiel an:

```
1 Prozedur rekursiveProzedur(Zahl)
2     Gib Zahl auf dem Bildschirm aus
3     Wenn (Zahl <= 0)
4         Gib den Text "Selbstaufruf−Stopp." auf dem Bildschirm aus
5     Sonst
6         rekursiveProzedur(Zahl − 1)
```

Diese rekursive Prozedur hat einen Parameter *Zahl*. Wenn sich die Prozedur selbst aufruft, übergibt sie einen Wert, der um eins kleiner. Wenn der übergebene Wert Null ist (oder kleiner), dann ruft sich die Prozedur nicht noch einmal selbst auf – das ist die Abbruchbedingung. Was passiert jetzt, wenn man *rekursiveProzedur*(2) startet? Zunächst einmal wird „2“ auf dem Bildschirm ausgegeben, anschließend ruft sich die Prozedur mit dem Wert 1 wieder auf, „1“ wird ausgegeben und das Programm ruft sich selbst mit dem Wert 0 auf. Auch „0“ wird ausgegeben, aber jetzt ruft sich die Prozedur nicht mehr selbst auf, sondern gibt einen Text aus. Die Ausgabe ist also

```
2 1 0 Selbstaufruf-Stopp.
```

Wenn man das Programm mit einer negativen Zahl aufruft, gibt es diese aus und stoppt dann direkt.

Zum Abschluss dieses kleinen Ausflugs geben wir noch ein drittes Beispiel, das einen weiteren Effekt veranschaulicht, den man bei rekursiven Programmen ausnutzt. Wenn ein Programm eine Prozedur aufruft, dann springt das Programm zu dieser Prozedur, führt diese aus und kehrt danach zu dem Punkt in der Programmausführung zurück, wo die Prozedur aufgerufen wurde. Das ist bei rekursiven Programmen genauso, aber die Effekte, die dadurch auftreten, sind komplizierter. Wir schauen uns dazu folgende Prozedur an:

```
1 Prozedur rekursiveProzedur(Zahl)
2     Gib Zahl auf dem Bildschirm aus
3     Wenn (Zahl <= 0)
4         Gib den Text "Selbstaufruf−Stopp." auf dem Bildschirm aus
5     Sonst
6         rekursiveProzedur(Zahl − 1)
7     Gib die Zahl (Zahl^2) auf dem Bildschirm aus.
```

Der Unterschied besteht nur in der neuen letzten Zeile. Diese Zeile wird nur erreicht, wenn der rekursive Aufruf (also der Aufruf, bei dem sich die Prozedur selbst aufruft) beendet ist. Wenn wir die Prozedur z.B. mit dem Wert 1 starten, so gibt die Prozedur „1“ aus, ruft sich mit dem Wert 0 auf, gibt „0“und den Text aus und dann das Quadrat von 0 (weil sie sich ja jetzt nicht selbst aufruft, sondern nach dem **Wenn**-Teil direkt weiterausführt). Anschließend ist der Aufruf mit dem Wert 0 beendet, so dass das Programm zurückspringt in den Aufruf mit dem Wert 1 und abschließend das Quadrat von 1 ausgibt. Ruft man die Prozedur mit dem Wert 5 aus, erhält man folgende Ausgabe:

```
5 4 3 2 1 0 Selbstaufruf-Stopp. 0 1 4 9 16 25
```

Alle Reihenfolgen durchprobieren

In Abschnitt haben wir uns überlegt, wie man ausrechnet, wie lange Hugo benötigt, wenn die Reihenfolge, in der er seine Kollegen besucht, feststeht.

Grundidee

Der erste Lösungsansatz, den wir besprechen wollen, beruht deshalb darauf, alle verschiedenen möglichen Besuchsreihenfolgen durchzugehen. Das machen wir rekursiv mit Hilfe einer Prozedur *Laufplan*. Wir besprechen die Idee anhand des folgenden Pseudocodes:

```
1  Prozedur Laufplan(verbleibendeMitarbeiter, Route)
2    Wenn (verbleibendeMitarbeiter leer) Dann
3      Berechne erforderlicheZeit als Dauer von Route
4      Wenn (erforderlicheZeit ≤ erlaubteZeit) Dann
5        Gib Route als Lösung aus
6        Beende alle laufenden Aufrufe von Laufplan
7      Sonst
8        Beende diesen Aufruf von Laufplan
9    Sonst
10     Wiederhole für jedes Element neuerMitarbeiter in der Liste verbleibendeMitarbeiter:
11       Verlängere Route um neuerMitarbeiter
12       Entferne neuerMitarbeiter aus verbleibendeMitarbeiter
13       Laufplan(verbleibendeMitarbeiter, Route)
14       Füge neuerMitarbeiter zu verbleibendeMitarbeiter hinzu
15       Verkürze Route um neuerMitarbeiter
```

Die Prozedur *Laufplan* erhält als Parameter die beiden Listen *verbleibendeMitarbeiter* und *Route*. Die Liste *Route* ist eine Liste einiger Mitarbeiter und stellt die Reihenfolge dar, in der Hugo diese besuchen möchte. In der Liste *verbleibendeMitarbeiter* sind alle Mitarbeiter, die in der Route noch nicht enthalten sind, d.h. diese müssen noch eingeplant werden. Die Variable *erlaubteZeit* wird außerhalb der Prozedur gesetzt und enthält das Zeitlimit, d.h. den letztmöglichen Zeitpunkt, zu dem Hugo noch einen Mitarbeiter treffen kann.

Wir schauen uns die Funktionsweise an einem Beispiel mit dem Mitarbeitern `Anna, Britta, Christine, Detlef` und `Eva` an. Dafür nehmen wir an, dass wir bereits mitten in der Ausführung sind und betrachten den Aufruf `Laufplan({Anna, Christine, Detlef},{Eva, Britta})`. `Eva` und `Britta` sollen also von Hugo zuerst aufgesucht werden und `Anna, Christine` und `Detlef` sind noch übrig. Weil *verbleibendeMitarbeiter* nicht leer ist, sondern noch drei Mitarbeiter enthält, werden die Zeilen 3 bis 8 übersprungen und die Prozedur springt in den `Sonst` -Teil ab Zeile 9. Dort findet sich eine Schleife, die für jedes Element in *verbleibendeMitarbeiter* einmal ausgeführt wird. Wir beginnen mit `Anna` und verlängern *Route* um `Anna` , d.h. die bisher geplante Besuchsreihenfolge ist nun `{Eva, Britta, Anna}` . `Anna` ist jetzt eingeplant, so dass *verbleibendeMitarbeiter* in Zeile 12 zu `{Christine, Detlef}` aktualisiert wird. Jetzt ruft sich die Prozedur selbst auf. Mit den aktuellen Werten der Variablen lautet der Aufruf `Laufplan({Christine, Detlef},{Eva, Britta, Anna})` . Bevor wir uns darum kümmern, was in diesem Aufruf weiter passiert, überlegen wir uns erst, wie es weitergeht, wenn das Programm nach der Ausführung des rekursiven Aufrufs hinter Zeile 13 weiterarbeitet. Dort werden die Variablen wieder in den Zustand gebracht, in dem sie vor der Ausführung von Zeile 11 und 12 waren. Anschließend folgt der nächste Schleifendurchlauf. Man kann sich überlegen, dass in diesem Durchlauf der Schleife der rekursive Aufruf `Laufplan({Anna, Detlef},{Eva, Britta, Christine})` erfolgt. Der dritte und letzte Durchlauf der Schleife bringt schließlich den rekursiven Aufruf `Laufplan({Anna, Christine},{Eva, Britta, Detlef})` . Wir stellen fest, dass die drei rekursiven Aufrufe genau die drei Möglichkeiten darstellen, wie man die Route `{Eva, Britta}` um einen der Mitarbeiter `{Anna, Christine, Detlef}` verlängern kann.

Und wie geht es weiter? Innerhalb des Aufrufs

```
Laufplan({Christine, Detlef},{Eva, Britta, Anna})
```

wird als erstes der rekursive Aufruf

```
Laufplan({Detlef},{Eva, Britta, Anna, Christine})
```

gestartet. Die Abarbeitung dieses Aufrufs wiederum startet:

```
Laufplan({}, {Eva, Britta, Anna, Christine, Detlef})
```

Bei diesem Aufruf werden jetzt die Zeilen 3 bis 7 statt der Zeilen 10 bis 15 ausgeführt. Zuerst wird wie in Abschnitt beschrieben berechnet, wie lange Hugo braucht um `Eva, Britta, Anna, Christine` und `Detlef` in dieser Reihenfolge zu besuchen. Anschließend wird getestet, ob diese Zeit klein genug ist. Wenn ja, haben wir eine Lösung gefunden. In diesem Fall beenden wir die Prozedur und *alle* anderen Aufrufe der Prozedur[2], denn wir brauchen ja nur eine Lösung zu suchen, die die Zeitschranke einhält (und nicht alle Lösungen oder die, die die wenigste Zeit braucht). Ist unsere Route keine Lösung, so beenden wir nur diesen rekursiven Aufruf und kehren in die Instant von *Laufplan* zurück, die uns aufgerufen hat. Dort ist die Schleife beendet, weil es nur `Detlef` in der Liste *verbleibendeMitarbeiter* gab. Wir kehren also noch eine Stufe zurück und rufen dort als nächstes `Laufplan({Christine},{Eva, Britta, Anna, Detlef})` auf. Durch einen weiteren rekursiven Aufruf kommen wir zu `Laufplan({},{Eva, Britta, Anna, Detlef, Christine})` und haben so eine weitere fertige Route, deren Länge nun bestimmt wird.

[2] Das kann man z.B. machen, indem man eine boolsche Variable einführt, die zwischen Zeile 13 und 14 abgefragt wird. Ist die Variable gesetzt, wird dort die Schleife abgebrochen. Auf diese Weise werden alle noch laufenden *Laufplan*-Aufrufe abgebrochen, sobald das Programm aus der Rekursion zurückkehrt.

Man kann sich überlegen, dass die Prozedur *Laufplan* **alle** möglichen Reihenfolgen von Mitarbeitern durchgeht – bis ggf. eine Lösung gefunden ist –, wenn man sie mit einer leeren Liste *Route* und einer alle Mitarbeiter enthaltenden Liste *verbleibendeMitarbeiter* startet.

Verbesserte Version

Der bisher beschriebene Ansatz probiert *alle* möglichen Reihenfolgen aus, in denen Hugo die Mitarbeiter besuchen kann. Damit findet er sicher eine Lösung, die die Zeitschranke einhält, wenn eine solche existiert. Andererseits dauert das aber auch ziemlich lange, weil es sehr viele mögliche Reihenfolgen gibt[3].

Bei Backtracking versucht man, viele Lösungen auf einmal auszuschließen, indem man früher erkennt, dass die bisher getroffenen Entscheidungen nicht mehr zum Ziel führen können. Bei unserem Problem ist das zum Beispiel der Fall, wenn wir eine Teilroute haben, die bereits das Zeitlimit überschreitet. Egal, wie wir die Route mit den verbleibenden Mitarbeitern fortsetzen, werden wir sicher keine gültige Lösung mehr finden. Deshalb kann man die Rekursion in diesem Fall abbrechen und spart sich so das Testen aller Möglichkeiten, die restlichen Mitarbeiter zu besuchen.

Damit das funktioniert, muss man die Länge der aktuellen Route immer mitberechnen, anstatt sie erst für vollständige Routen zu berechnen. Macht man das, kann man vor jedem rekursiven Aufruf testen, ob das Zeitlimit bereichts überschritten ist. Der folgende Pseudocode setzt diese Idee in die Tat um:

```
1  Prozedur Laufplan(verbleibendeMitarbeiter, Route, aktuellerOrt, aktuelleZeit)
2    Wenn (verbleibendeMitarbeiter leer) Dann
3      Wenn (aktuelleZeit ≤ erlaubteZeit) Dann
4        Gib Route als Lösung aus
5        Beende alle laufenden Aufrufe von Laufplan
6      Sonst
7        Beende diesen Aufruf von Laufplan
8    Sonst
9      Wiederhole für jedes Element neuerMitarbeiter in der Liste verbleibendeMitarbeiter:
10       Setze neuerOrt auf die Teeküche von neuerMitarbeiter
11       Berechne Laufzeit als die notwendige Zeit, um von aktuellerOrt zu neuerOrt zu gelangen
12       Setze Ankunftszeit auf (aktuelleZeit + Laufzeit)
13       Berechne neueZeit als kleinste Zeit ≥ Ankunftszeit, zu der neuerMitarbeiter Tee trinkt
14       Wenn (neueZeit ≤ erlaubteZeit) Dann
15         Verlängere Route um neuerMitarbeiter
16         Entferne neuerMitarbeiter aus verbleibendeMitarbeiter
17         Laufplan(verbleibendeMitarbeiter, Route, neuerOrt, neueZeit)
18         Füge neuerMitarbeiter zu verbleibendeMitarbeiter hinzu
19         Verkürze Route um neuerMitarbeiter
20       Sonst
21         Beende diesen Aufruf von Laufplan
```

[3] Wenn es n Mitarbeiter gibt, gibt es auch n Möglichkeiten für Hugo, wen er als erstes besucht. Für den zweiten in der Route gibt es dann noch $n-1$, für den dritten $n-2$ Möglichkeiten usw., für den Mitarbeiter, den er zuletzt besucht, gibt es nur noch eine. Deshalb gibt es insgesamt $n \cdot n-1 \cdot n-2 \cdot \ldots \cdot 2 \cdot 1 = n!$ Möglichkeiten für Hugo. Diese Zahl nennt man die *Fakultät* von n. Für $n = 5$ ist die Fakultät von n bereits 120, für $n = 7$ ist sie 5040 und für $n = 10$ erhält man 3628800.

Die Parameter *verbleibendeMitarbeiter* und *Route* bedeuten weiterhin das gleiche und werden auch genauso aktualisiert wie bisher. *aktuellerOrt* gibt den Ort an, an dem Hugo nach Abarbeitung der bisherigen Route stehen würde[4]. Am Anfang ist *aktuellerOrt* 1, weil Hugo in Teeküche 1 startet. Der Parameter *aktuelleZeit* gibt an, wie viel Zeit man für das Geldeinsammeln auf der bisher gespeicherten *Route* benötigt. Beim ersten Aufruf von *Laufplan* ist dieser Parameter 0.

Die Zeilen 10 bis 12 sind neu. Hier wird berechnet, wie viel zusätzliche Zeit Hugo benötigt, wenn er nach Abarbeitung von *Route* als nächstes *neuerMitarbeiter* besuchen möchte. Dazu muss er zuerst in dessen Teeküche *neuerOrt* laufen. Die dafür benötigte Zeit wird berechnet und in *Laufzeit* gespeichert. Anschließend muss Hugo noch warten, bis der Mitarbeiter in der Teeküche erscheint, so dass er das Geld von *neuerMitarbeiter* schließlich zum Zeitpunkt *neueZeit* einsammelt.

Da wir jetzt wissen, wie lange die bisherige Route einschließlich dem Besuch bei *neuerMitarbeiter* dauert, können wir in Zeile 14 eine Abfrage einfügen, die verhindert, dass wir einen rekursiven Aufruf machen, obwohl die bisherige Route bereits zu lange dauert.

Durch diese Verbesserung ändert sich im schlechtesten Fall nichts an der Anzahl Möglichkeiten, die wir durchprobieren müssen. Das ist beim Backtracking oft so: Man versucht, den Ansatz praktisch schneller zu machen, ohne eine Garantie auf Beschleunigung zu haben.

Weitere Verbesserungen

In den letzten beiden Abschnitten haben wir ein Grundgerüst für eine Lösung mit Backtracking beschrieben. Trotzdem löst das Programm nicht alle vorgegebenen Beispiele innerhalb vernünftiger Zeit. Es sind weitere Verbesserungen denkbar, die das Programm weiter beschleunigen. Da wir nur *irgendeine* Lösung benötigen, die das Zeitlimit einhält, ist es besonders lukrativ zu versuchen, möglichst schnell eine möglichst gute Lösung zu finden (und zu hoffen, dass diese gut genug ist). Dazu kann man z.B. die Reihenfolge, in der man die Möglichkeiten durchprobiert, verändert. Bisher ist die Reihenfolge, in der die Mitarbeiter in der ursprünglichen Liste stehen, entscheidend dafür, in welcher Reihenfolge die verschiedenen Besuchsreihenfolgen getestet werden. Stattdessen kann man auch bei jedem Aufruf der Prozedur *Laufplan* berechnen, zu welchen Zeitpunkten man die verbleibenden Mitarbeiter jeweils treffen kann und die Liste der verbleibenden Mitarbeiter vor Ausführung der Schleife in Zeile 9 absteigend nach den berechneten Zeiten sortieren. Alternativ kann man testen, welchen Einfluss die Teezeiten haben und die Liste z.B. aufsteigend oder absteigend nach Teezeiten sortieren.

Wir wollen diese Ansätze nicht weiter verfolgen, sondern uns zum Abschluss einem grundsätzlich anderen Ansatz mit Backtracking widmen.

Alle Zeitpunkte durchprobieren

Bei unserem ersten Ansatz entscheiden wir uns nach und nach für die Mitarbeiter, die wir in einer bestimmten Reihenfolge besuchen wollen. Das machen wir jetzt anders. Zuerst sortieren

[4] Diese Variable ist nicht zwingend nötig, weil man den letzten Ort auch aus dem Ort des letzten Mitarbeiters aus *Route* ermitteln kann, aber die Verwendung von *aktuellerOrt* macht den Pseudocode übersichtlicher.

wir die Mitarbeiter absteigend nach ihren Teezahlen und behalten diese Sortierung während des gesamten Programms. Wir probieren dabei nicht alle Reihenfolgen durch, sondern testen für jeden Mitarbeiter die möglichen Zeitpunkte, zu denen man diesen besuchen kann. Die Reihenfolge der Mitarbeiter ergibt sich dann implizit. Zur Veranschaulichung betrachten wir zunächst ein Beispiel.

Nehmen wir an, wir hätten die Mitarbeiter `Felix, Jessica` und `Vanessa` mit den Teezeiten 10, 40 und 15 bei einem Zeitlimit von 60 Minuten. Dann sortieren wir die Mitarbeiter zuerst in der Reihenfolge `Jessica (40), Vanessa (15)` und `Felix (10)` . Jetzt beginnen wir unsere Rekursion bei `Jessica` . Es gibt nur eine Möglichkeit, Jessica innerhalb des Limits zu besuchen, nämlich zum Zeitpunkt 40. Wir probieren also den Zeitpunkt 40 aus und testen zuerst, ob das ohne Berücksichtigung weiterer Mitarbeiter geht. Dazu müssen wir nur überprüfen, ob man die Teeküche von `Jessica` von Teeküche 1 aus innerhalb von 40 Minuten erreichen kann. Das Beispiel soll so gewählt sein, dass das funktioniert.

Jetzt machen wir einen rekursiven Aufruf und möchten als nächstes einen Besuchszeitpunkt für `Vanessa` finden. Wir legen fest, dass das Beispiel so ist, dass der Weg zwischen Teeküche 1 und `Vanessa` s Teeküche 16 Minuten dauert und der Weg zwischen der Teeküche von Jessica und der von `Vanessa` 6 Minuten. Wir kümmern uns nicht darum, ob man so eine Situation durch eine geschickte Verteilung der Teeküchen erreichen kann, es geht nur darum, dass wir Zahlen für unser Beispiel haben. Wir könnten `Vanessa` zu den Zeitpunkten 15, 30, 45 und 60 besuchen. Wir testen zuerst, welche Zeitpunkte noch möglich sind. Der Zeitpunkt 15 ist nicht möglich, weil Hugo `Vanessa` s Teeküche in 15 Minuten nicht erreichen kann (weil er in Teeküche 1 startet). Der Zeitpunkt 45 ist ebenfalls nicht möglich, weil wir ja zum Zeitpunkt 40 noch bei `Jessica` sind und in den Weg zu `Vanessa` s Teeküche in 5 Minuten nicht zurücklegen könnne. Es bleiben also die Zeitpunkte 30 und 60, gegen die nichts spricht, und wir probieren zuerst den Zeitpunkt 30. In der nächsten Stufe der Rekursion würden wir nun versuchen, `Felix` noch im Plan unterzubringen. Bisher besuchen wir `Vanessa` nach 30 Minuten und `Jessica` nach 40 Minuten. `Felix` könnte man prinzipiell zu den Zeitpunkten 10, 20, 30, 40, 50 und 60 besuchen. Welche davon möglich sind, wollen wir uns jetzt nicht weiter überlegen (dafür müssten wir noch mehr Annahmen über die Abstände zwischen den Teeküchen treffen). Wenn wir einen Zeitpunkt finden, zu dem wir `Felix` besuchen können, dann haben wir eine Lösung gefunden, die die Zeitschranke einhält (weil wir gar keine anderen Lösungen betrachten). Wenn wir `Felix` in unserem bisherigen Plan nicht unterbringen können, gehen wir eine Rekursionsstufe zurück und besuchen `Vanessa` nach 60 statt nach 30 Minuten und versuchen erneut, `Felix` unterzubringen.

Diese Vorgehensweise ist auf den ersten Blick komplizierter als unser erster Ansatz. Wir besprechen ihn genauer anhand des folgenden Pseudocodes:

```
1  Prozedur Laufplan(Zeitplanung, Zuordnung, aktuellerMitarbeiter)
2    Wenn (aktuellerMitarbeiter = letzterMitarbeiter) Dann
3      Gib die Zeitpunkte in Zeitplanung und die zugehörigen Mitarbeiter als Lösung aus
4      Beende alle laufenden Aufrufe von Laufplan
5    Sonst
6      Setze neuerMitarbeiter auf den Nachfolger von aktuellerMitarbeiter in sortierteListe
7      Setze neueTeezahl auf die Teezahl von neuerMitarbeiter
8      Setze neuerOrt auf die Teeküche von neuerMitarbeiter
9      Wiederhole für alle Vielfachen neueZeit von neueTeezahl, die kleiner als erlaubteZeit sind:
10
11       Wenn (ZeitDavor enthalten in Zeitplanung) Dann
12         Setze ZeitDavor auf den Vorgänger von neueZeit in Zeitplanung
13         Setze MitarbeiterDavor auf den Mitarbeiter, der laut Zuordnung für ZeitDavor geplant
             ist
14         Setze OrtDavor auf die Teeküche von MitarbeiterDavor
15       Sonst
16         Setze ZeitDavor auf 0
17         Setze MitarbeiterDavor auf Hugo
18         Setze OrtDavor auf 1
19
20       Berechne notwendigeZeit als die Zeit für den Weg von OrtDavor nach neuerOrt
21
22       Wenn (notwendigeZeit+ZeitDavor ≤ neueZeit) Dann
23         Setze ersterTest auf erfolgreich
24       Sonst
25         Setze ersterTest auf erfolglos
26
27       Wenn (ZeitDanach enthalten in Zeitplanung) Dann
28         Setze ZeitDanach auf den Nachfolger von neueZeit in Zeitplanung
29         Setze MitarbeiterDanach auf den Mitarbeiter, der laut Zuordnung für ZeitDanach
             geplant ist
30         Setze OrtDavor auf die Teeküche von MitarbeiterDavor
31         Berechne notwendigeZeit als die Zeit für den Weg von neuerOrt nach OrtDanach
32
33         Wenn (neueZeit+notwendigeZeit ≤ erlaubteZeit) Dann
34           Setze zweiterTest auf erfolgreich
35         Sonst
36           Setze zweiterTest auf erfolglos
37       Sonst
38         Setze zweiterTest auf erfolgreich
39
40       Wenn (ersterTest erfolgreich und zweiterTest erfolgreich) Dann
41           Trage neueZeit in Zeitplanung ein
42           Speichere, dass zum Zeitpunkt neueZeit neuerMitarbeiter besucht wird
43           Laufplan(Zeitplanung, neuerMitarbeiter)
44           Entferne die Verbindung zwischen neueZeit und neuerMitarbeiter
45           Entferne neueZeit aus Zeitplanung
46       Sonst
47         Beende diesen Aufruf von Laufplan
```


Parameter und globale Variablen Die Prozedur erhält die Parameter *Zeitplanung*, *Zuordnung*und *aktuellerMitarbeiter*. Der Parameter *aktuellerMitarbeiter* gibt den Mitarbeiter an, den Hugo als letztes besucht hat. Am Anfang kann man zum Beispiel Hugo selbst als letzten Mitarbeiter übergeben.

Die beiden anderen Parameter sind von einem komplizierteren Typ. In *Zeitplanung* speichern wir die Zeitpunkte ab, für die wir bereits ein Treffen eingeplant haben. Die Datenstruktur dafür

kann verschieden aussehen, solange sie folgende Dinge unterstützt: Zunächst einmal möchten wir Zeitpunkte einfügen und löschen können. Außerdem soll uns die Datenstruktur zu der Zahl t den größten gespeicherten Zeitpunkt sagen, der kleiner ist als t. Diesen Zeitpunkt nennen wir Vorgänger. Analog soll uns die Datenstruktur den kleinsten gespeicherten Zeitpunkt sagen können, der größer als eine Zahl t ist – ihren Nachfolger. Man kann eine solche Datenstruktur als lineare Liste oder Array realisieren, effizienter ist ein Suchbaum[5] Wir übergeben dem ersten Prozeduraufruf eine leere *Zeitplanung*.

Die Variable *Zuordnung* enthält eine Zuordnung von Zeiten zu Mitarbeitern, d.h. sie kann zu einem Zeitpunkt, sofern der in der Zuordnung abgespeichert ist, den zugehörigen Mitarbeiter zurückliefern. Außerdem kann sie neue Paare aus Zeitpunkten und Mitarbeitern speichern oder gespeicherte Paare löschen. Auch die hierfür erforderliche Datenstruktur lässt sich durch eine Liste realisieren. Alternativ kann man Suchbäume oder Hashing verwenden.

Neben den Parametern verwendet die Prozedur wie im letzten Abschnitt die Variable *erlaubteZeit*, die den Zeithorizont enthält, und zusätzlich die Liste *sortierteListe*, die alle Mitarbeiter enthält, absteigend sortiert nach ihren Teezahlen.

Ausführung der Prozedur Nehmen wir an, wir befinden uns mitten in der Rekursion und haben bereits einige Treffen geplant, andere noch nicht. In diesem Fall enthält *Zeitplanung* bereits einige Zeitpunkte und *Zuordnung* die entsprechenden Mitarbeiter, die wir treffen möchten, aber *aktuellerMitarbeiter* ist noch nicht *letzterMitarbeiter*. Wir führen also den Code ab Zeile 6 aus. Zuerst wird der nächste Mitarbeiter ermittelt (das ist einfach der mit der nächstkleineren Teezahl) und seine Teezahl sowie seine Teeküche werden gespeichert. Anschließend testet die Schleife in Zeile 9 alle Zeitpunkte durch, zu denen man den Mitarbeiter treffen kann (das sind genau die echt positiven Vielfachen der Teezahl des Mitarbeiters).

Für einen möglichen Treffzeitpunkt wird dann geprüft, ob man dieses Treffen in den bisherigen Plan einfügen kann. Dazu wird ermittelt, wo sich Hugo vor und nach diesem Treffen aufhalten muss. Die Zeit, die Hugo braucht, um die Strecken zu dem neuen Treffpunkt und von diesem weg zurückzulegen, müssen klein genug sein, um zwischen den bereits geplanten Treffen eingebaut werden zu können. Dabei gibt es zwei Ausnahmen: Wenn vor dem neuen Treffzeitpunkt noch kein Treffen eingeplant ist, läuft Hugo direkt von seinem Startort, Teeküche 1, zu dem neuen Treffpunkt. Wenn nach dem neuen Zeitpunkt noch kein Treffen eingeplant ist, kann Hugo seine Tour einfach dort beenden, so dass hier nichts getestet werden muss.

Wenn beide Tests erfolgreich waren, trägt die Prozedur den neuen Zeitpunkt in *Zeitplanung* und das Paar aus Zeitpunkt und Mitarbeiter in *Zuordnung* ein. Anschließend ruft sich die Prozedur selbst auf und legt so die Treffen mit den weiteren Mitarbeitern fest.

Vorteile Im schlechtesten Fall ist der gerade beschriebene Ansatz nicht besser als der erste[6]. Trotzdem funktioniert er für viele Beispiele deutlich besser (u.a. auf den vorgegebenen).

[5] Eine allgemeine Einführung in Suchbäume und das später erwähnte Hashing gibt es z. B. in Kapitel 3 von http://ls2-www.cs.uni-dortmund.de/lehre/sommer2007/dap2/skript.pdf.

[6] Eventuell sogar eher schlechter. Wenn es n Mitarbeiter mit den Teezahlen $t_1, \ldots, t_n$ gibt und das Zeitlimit k ist, so kann man sich überlegen, dass wir bis zu $\prod_{i=1}^{n} \frac{k}{t_i}$ Möglichkeiten durchprobieren müssen. Bei kleinen Teezahlen ist diese Zahl riesig.

Durch die absteigende Sortierung der Mitarbeiter nach Teezahlen erreichen wir, dass Mitarbeiter, die wir nur schwer treffen können, sicher eingeplant werden. Die Chance, Mitarbeiter mit kleinen Teezahlen weiter hinten in der Rekursion noch irgendwo in den Zeitplan „hineinquetschen“zu können, ist groß. Wenn alle Teezahlen im Vergleich zum Zeitlimit eher klein sind, könnte der Ansatz prinzipiell in Schwierigkeiten geraten, weil es bereits auf der ersten Stufe viele verschiedene Möglichkeiten gibt. Bei sehr kleinen Teezahlen ist es aber meistens auch einfach, eine gültige Lösung zu finden, ohne viele Möglichkeiten auszuprobieren. Wirklich unangenehm wird es erst, wenn die Teezahlen im Vergleich zum Zeitlimit eher klein sind *und* die Anzahl der Teeküchen (und damit die Abstände zwischen einigen Teeküchen) so groß ist, dass man trotz der kleinen Teezahlen innerhalb des Zeitlimits nicht alle Mitarbeiter besuchen kann. In diesem Fall sollte man eher auf den ersten Ansatz zurückgreifen.

Lösungen für die vorgegebenen Beispiele

Für die Beispiele 2 und 3 gibt es keine Möglichkeit, die Mitarbeiter im vorgegebenen Limit zu besuchen. Für die anderen Beispiele werden Lösungen angegeben; dabei kann es auch andere gültige Laufwege geben, auch mit anderer Zeitdauer.

Beispiel 0

Eingabe: *(Zeitlimit 240 Minuten)*	(Einzige) Lösung:
`14400 8`	`0 Hugo`
`Daniel 120 3`	`121 Daniela`
`Daniela 121 7`	`240 Daniel`

Hugo trifft also zuerst Daniela nach 121 Minuten und dann Daniel nach 240 Minuten.

Beispiel 1

Eingabe: *(Zeitlimit 120 Minuten)*	Lösung:
`7200 8`	`0 Hugo`
`Johanna 60 2`	`58 Maja`
`Elias 82 2`	`59 Sophia`
`Maja 58 6`	`60 Johanna`
`Philipp 58 8`	`60 Alina`
`Lina 77 2`	`61 Nico`
`Tom 75 5`	`75 Tom`
`Alina 60 2`	`77 Lina`
`Nico 61 5`	`78 Nils`
`Sophia 59 3`	`82 Elias`
`Nils 78 5`	`116 Philipp`

Beispiel 4

Eingabe: *(Zeitlimit 420 Minuten)*

```
25200 256
Hanna 76 147
Leon 73 127
Lena 85 166
Luca 164 82
Lea 82 207
Tim 105 13
Mia 277 139
Jonas 34 243
Lilli 89 205
Maximilian 66 200
Sarah 32 15
Max 190 182
Neele 42 221
Niklas 72 81
Sophie 137 94
Ben 79 209
Julia 42 140
Jannik 53 26
Lisa 48 184
Noah 43 146
Amelie 31 97
Moritz 54 169
Leni 33 139
David 33 128
Luisa 31 42
Simon 95 134
Clara 35 94
Erik 96 244
Josephine 79 68
Alexander 47 21
```

Lösung:

```
0 Hugo
31 Luisa
47 Alexander
53 Jannik
64 Sarah
84 Neele
96 Erik
105 Tim
137 Sophie
144 Niklas
158 Josephine
164 Luca
175 Clara
186 Amelie
198 David
216 Moritz
237 Ben
255 Lena
264 Leni
277 Mia
285 Simon
292 Leon
301 Noah
304 Hanna
328 Lea
340 Jonas
356 Lilli
380 Max
384 Lisa
396 Maximilian
420 Julia
```


Beispiel 5

Eingabe: *(Zeitlimit 720 Minuten)*

```
43200 256
Jana 59 53
Florian 121 54
Annika 51 188
Linus 52 181
Jasmin 106 158
Jason 99 106
Finja 57 12
Lennard 61 104
Katharina 55 153
Jannis 108 24
Fiona 72 38
Dominic 128 225
Viktoria 50 255
Marc 91 131
Vanessa 117 82
Jonathan 69 244
Emelie 86 95
Colin 61 108
Isabell 77 110
Kevin 119 229
Nina 64 222
Robin 47 105
Lucy 69 8
Adrian 54 160
Maria 76 143
Henry 134 15
Caroline 71 52
Aaron 52 66
Selina 84 124
Matthis 102 199
Helena 75 98
Sebastian 71 102
Kim 89 225
Raphael 113 121
Mara 85 238
Maurice 59 184
Melissa 75 191
Jonah 122 212
Pauline 62 36
Maxim 52 42
Amy 156 154
Lasse 120 162
Sina 92 219
Kilian 63 44
Theresa 96 228
Karl 60 197
Elena 55 183
Leonard 83 35
Lotta 69 197
Oskar 54 85
```

Lösung:

```
0 Hugo
50 Viktoria
57 Finja
69 Jonathan
85 Mara
92 Sina
102 Matthis
110 Elena
122 Jonah
128 Dominic
138 Lotta
156 Amy
162 Adrian
165 Katharina
177 Maurice
192 Theresa
207 Lucy
216 Jannis
236 Jana
242 Florian
248 Pauline
249 Leonard
252 Kilian
268 Henry
300 Karl
306 Annika
318 Jasmin
336 Selina
339 Raphael
360 Lasse
380 Maria
396 Jason
423 Robin
426 Sebastian
427 Lennard
430 Emelie
476 Kevin
520 Linus
525 Melissa
546 Marc
572 Aaron
585 Vanessa
594 Oskar
600 Helena
610 Colin
616 Isabell
639 Caroline
648 Fiona
676 Maxim
704 Nina
712 Kim
```

Bewertungskriterien

- Bei der Berechnung von Lösungen gibt es reichlich Möglichkeiten, Fehler zu machen oder sich durch Nichtbeachtung das Leben zu vereinfachen:
  - Die Laufzeit zwischen Teeküchen muss beachtet und korrekt berechnet werden.
  - Hugo muss auf die Mitarbeiter warten können, die Wartezeit muss korrekt berechnet werden.
  - Probleme bei gleichen Teeküchen: Wenn zwei Mitarbeiter die gleiche Teeküche haben, benötigt es keine Zeit, von einem zum nächsten zu gelangen.
  - Das Lösungsverfahren bzw. seine Implementation enthält grundlegende Fehler.
  - Das in Sekunden angegebene Zeitlimit wurde als Minutenwert interpretiert.

  Die Fehler können unterschiedlich schwerwiegend sein, und Nichtbeachtung eines Aspekts ist schlechter als ihn fehlerhaft zu behandeln.
- Das Lösungsverfahren darf nicht zu ineffizient sein. Auch hier ist eine Lösung nach dem „brute force“ Prinzip nicht geeignet. Entscheidend ist, dass in der Einsendung Verbesserungsmöglichkeiten erkannt und umgesetzt werden. Gute Hinweise auf die Effizienz der Lösung liefern die Beispiele 4 und 5: Beispiel 4 ist schwierig genug, um nur von ausgefeilteren Programmen lösbar zu sein. Beispiel 5 ist schon recht schwierig und muss nicht unbedingt gelöst worden sein. Fehlen beide ohne anderen ersichtlichen Grund, ist die Lösung ziemlich sicher zu ineffizient.
- Die Ausgaben für die vorgegebenen und für eigene Beispiele sollen korrekt sein: Lösungen sollen also genau für die Beispiele gefunden werden, für die Lösungen existieren. Natürlich sollen ausgegebene Lösungen auch in sich fehlerfrei sein.
- Zu allen vorgegebenen Beispielen müssen Ausgaben dokumentiert sein. Außerdem soll ein Programm auch mit selbst gewählten Eingaben getestet worden sein, die idealerweise Aufschluss über die Details von Verfahren und Implementierung liefern.

Aufgabe 5: Bürgerampel

Teilaufgabe 1 – Diagramm

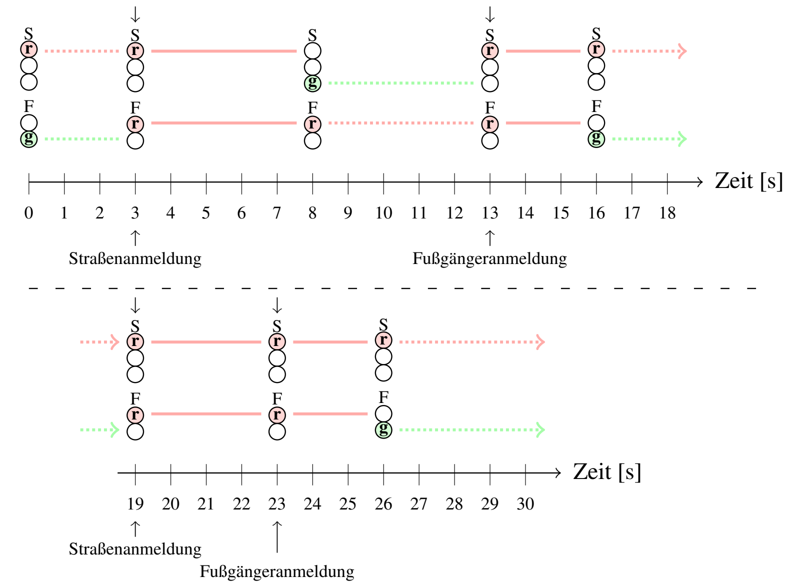


Abbildung 2: zeitlicher Ablauf der Ampelfunktion
Legende: S: Straßenampel, F: Fußgängerampel, die gestrichelten Balken geben frei wählbare Zeitdauern an.

In Abbildung 2 ist beispielhaft der zeitliche Ablauf der Ampelschaltung dargestellt: Am Anfang ist die Ampel für die Fußgänger grün. Nach 3 Sekunden fährt ein Fahrzeug auf die Induktionsschleife. Die (Fußgänger-) Ampel schaltet ganz auf rot und gibt den Fußgängern 5 Sekunden Zeit die Straße zu verlassen. Ab der achten Sekunde haben dann die Fahrzeuge grün – aber nicht lange, denn in der dreizehnten Sekunde will ein Pedener (so heißen nämlich die Bürger von Pedes) die Straße überqueren und hat dafür den Knopf betätigt. Die Ampel reagiert prompt und schaltet nach der dritten Sekunde Räumzeit auf Fußgänger-Grün. Das nächste Fahrzeug naht und wird in der neunzehnten Sekunde von der Ampel registriert. Noch während der Räumzeit drückt aber ein Fußgänger seinen Knopf und macht dem Fahrzeuglenker alle Hoffnung zunichte, grün zu bekommen. Denn nach der vorgeschriebenen Räumzeit haben nun die Fußgänger grün. . .

Teilaufgabe 2 – Testen

Alle Implementierungen erfüllen den einfachen Test, ob Anmelden bei jeweils grüner Ampel wirkungslos bleibt. Die Vorbedingungen für die Vorschriften 1 und 2 werden also immer beachtet.

Ebenfalls werden glücklicherweise in allen Implementierungen die Räumungszeiten eingehalten, also die Zeiten, bei denen beide Ampeln rot sind, damit Fußgänger, welche bei grün losgelaufen sind, noch die andere Straßenseite erreichen bzw. Fahrzeuge den Weg für Fußgänger frei machen können. Zu sicherheitskritischen Situationen dürfte es also bei keiner Implementierung kommen.

Dazugehörige Testfälle (bei allen Implementierungen in Ordnung):

Testfälle I: Räumzeiten eingehalten?

a) $\mathsf{Sg} \xrightarrow{\mathsf{FAn}} \mathsf{Sr} \xrightarrow{3s} \mathsf{Fg}$

b) $\mathsf{Fg} \xrightarrow{\mathsf{SAn}} \mathsf{Fr} \xrightarrow{5s} \mathsf{Sg}$

Erklärung für Kurzschreibweise bei den Testfällen

Sg/Sr Straßenampel grün bzw. rot

Fg/Fr Fußgängerampel grün bzw. rot

SAn Straßenanmeldung

FAn Fußgängeranmeldung

Die durch die Pfeile gekennzeichnete Schritte sind ohne längere Pausen auszuführen. Wenn keine explizite Pause angegeben ist, so muss der im jeweiligen Testfall folgende „Zustand“ sofort eintreten. Der erste Zustand in jedem Testfall ist die Startbedingung, welche vor Testbeginn manuell so eingestellt werden muss, dass sie – sofern möglich – statisch ist, sich also nicht automatisch ändert.

Die Vorschriften selber sagen nichts darüber aus, ob die jeweilige Ampel grün bleiben soll, solange keine Anmeldung durch den anderen Verkehrsteilnehmer vorliegt. Da aber die Ampel mit einer Bedarfsschaltung versehen werden soll und die Fußgänger gleichberechtigt sein sollen, ist naheliegend, dass die Ampel auch nur auf Bedarf reagieren soll. Außer Implementierung 1 verharren alle Implementierungen in der jeweiligen Grünphase. Implementierung 1 gibt dagegen den Fußgängern maximal 10 Sekunden um die Straße zu überqueren. Danach schaltet sie automatisch nach der vorgeschriebenen Räumzeit den Strassenverkehr frei. Es ist also Testfall IIa nicht erfüllt. Implementierung 1 setzt also eine „herkömmliche“ Fußgängerampel um, was bei geringem Fußgängeraufkommen durchaus sinnvoll sein kann.

Testfälle II: keine selbständigen Aktionen?

a) $\mathsf{Fg} \xrightarrow{\text{bleibt}} \mathsf{Fg}$

b) $\mathsf{Sg} \xrightarrow{\text{bleibt}} \mathsf{Sg}$

Weiterhin muss nach Vorschrift 3 der Schaltungsablauf immer neu gestartet werden, falls Bedarf angemeldet wird. Laut Spezifikation muss sich dadurch die Ampel unglücklicherweise

blockieren lassen. Dies alles lässt sich mit folgenden Testfällen überprüfen:

Testfälle III: Vorschrift 3 eingehalten?

a) $\mathsf{Sr} \xrightarrow{\mathsf{SAn}} \mathsf{Fr} \xrightarrow{\mathsf{FAn}} \xrightarrow{3s} \mathsf{Fg}$

b) $\mathsf{Fr} \xrightarrow{\mathsf{FAn}} \mathsf{Sr} \xrightarrow{\mathsf{SAn}} \xrightarrow{5s} \mathsf{Sg}$

c) $\mathsf{Fr} \xrightarrow{\mathsf{FAn}} \mathsf{Sr} \xrightarrow{\mathsf{FAn}} \cdots \xrightarrow{\mathsf{FAn}} \xrightarrow{3s} \mathsf{Fg}$

Diese Testfälle erfüllen Implementierung 1 und 3. In Implementierung 3 ist sogar die Ampel nach Ausführen des Testfalls IIIa in Fußgänger-Grün gefangen.

In Implementierung 2 können Fußgänger eine laufende Anmeldung nicht abbrechen bzw. neu starten. Eine Straßenanmeldung wird bei laufender Fußgängeranmeldung zwischengespeichert und sofort ausgeführt, wenn die Fußgänger grün erhalten. Eine Fußgängeranmeldung wird dagegen bei laufender Straßenanmeldung einfach ignoriert.

Implementierung 4 erfüllt diese Testfälle ebenfalls nicht. Bei ihr sind Mindestzeiten eingebaut, welche in der Spezifikation nicht genannt werden. Die einzelnen Schritte werden so erst gar nicht alle ausgeführt (Test IIIa) oder erst nach deutlicher Verzögerung umgesetzt (Test IIIb).

Eine Zusammenfassung ist in der Tabelle 3 dargestellt.

Tabelle 3: Zusammenfassung Testfälle

Testfall	**Implementierung**			
	1	**2**	**3**	**4**
Ia	ok	ok	ok	ok
Ib	ok	ok	ok	ok
IIa	nein	ok	ok	ok
IIb	ok	ok	ok	ok
IIIa	ok	nein	ok	nein
IIIb	ok	nein	ok	nein
IIIc	ok	nein	ok	nein

Die Implementierungen 2 und 4 zeigen also von den Vorschriften abweichendes Verhalten, die Spezifikationstreue von Implementierung 1 ist strittig. Dagegen sind in Implementierung 3 die Vorschriften genau umgesetzt. Da, wie unten beschrieben, die Systemspezifikation nicht sehr sinnvoll ist, sind die von ihr abweichenden Implementierungen für den Alltag aber tauglicher.

Teilaufgabe 3 – Systemspezifikation

Die Systemspezifikation weist erhebliche Mängel auf:

- Durch wiederholtes Anmelden von Fußgängerbedarf bei roter Fußgängerampel kann der Verkehr blockiert werden. Die Vorschrift 1 wird dann nach der Spezifikation immer wieder von vorne ausgeführt. Sowohl Fußgängerampel als auch Straßenampel bleiben rot, sofern die Fußgänger häufiger als alle 3 Sekunden auf einen ihrer Knöpfe drücken. Dieses Problem tritt bei den Implementierungen 1 und 3 auf.
- Falls nach Meldung von Straßenbedarf Fußgänger innerhalb von 5 Sekunden ihrerseits Bedarf anmelden, gerät die Straßenanmeldung in Vergessenheit. Die Fußgänger haben darauf immer grün, da die Autofahrer ihren Bedarf nur beim Überfahren, aber nicht beim unbewegten Stehen vor der Ampel anmelden können. Die Autofahrer müssten also umkehren (was bei viel Verkehr recht schwierig sein dürfte) und nochmal über die Induktionsschleife fahren, um wieder eine Chance auf Grün zu erhalten. Dieses Problem tritt nur in Implementierung 3 auf.
- Dadurch, dass nahezu ständig zwischen Grün für Fußgänger und Fahrzeuge umgeschaltet werden kann, summieren sich die Räumzeiten und verhindern flüssigen Verkehr.
- Es ist nicht geregelt, was passiert, wenn Straßen- und Fußgängerbedarf bei roter Fußgänger- und Straßenampel *gleichzeitig* angemeldet werden.
- Wenn die Fahrzeuge in einem bestimmten Abstand fahren, kann es leicht passieren, dass die Fußgänger mehrmals drücken müssen, um grün zu bekommen, da die Fußgängeranmeldungsvorgang durch ein neu auf die Induktionsschleife fahrendes Fahrzeug unterbrochen wird.
- Es ist nicht geregelt, was passiert, wenn die Ampel das erste Mal eingeschaltet wird.
- Die Abschaffung der Gelbphase führt wahrscheinlich zu einer Reihe von Auffahrunfällen, da Fahrzeuge plötzlich stark bremsen müssen (weil etwa eine Ampel ohne Vorwarnung plötzlich rot wird). Um diesem Problem zu entgehen, müssten die Fahrzeuge einen sehr viel größeren Sicherheitsabstand einhalten, wodurch aber der Durchsatz der Straßen verringert wird.
- Ob die Räumzeiten (3 s bzw. 5 s) ausreichen, ist fraglich.

Aus all diesem ergibt sich, dass Fußgänger und Straßenfahrzeuge, anders als von den Bürgern von Pedes beabsichtigt, nicht gleichberechtigt sind. Die Fußgänger können die Ampel blockieren oder sich nahezu ständiges Grün sichern.

Die meisten der beschriebenen Probleme lassen sich durch die Einführung von Mindestzeiten für die Grünphasen lösen. Dass jederzeit ein Schaltungsvorgang abgebrochen werden kann (Vorschrift 3), erscheint nicht sinnvoll, da dadurch die Ampel blockiert werden kann und unnötig oft Räumzeiten erforderlich sind.

Bewertungskriterien

- Teil 1: Im Diagramm sollte für eine gewisse Zeit die Ampelfunktion korrekt dargestellt werden: Es sollte mindestens für eine Straßenbedarfs- und Fußgängerbedarfsanmeldung der zeitliche Ablauf der jeweiligen Signalwechsel aufgezeigt werden. Die Hauptachse soll dabei die Zeitachse sein, die Zeitpunkte der Bedarfsanmeldungen und der Farbwechsel der Ampeln soll erkennbar sein.

  Ein allgemeines Zustandsdiagramm, das die Ampelspezifikation wiedergibt, erfüllt die Aufgabe nicht.
- Teil 2: Es sollten folgende Besonderheiten erkannt werden, welche von der Spezifikation abweichen bzw. nicht erwähnt werden.
  - Implementierung 1: Die Ampel begrenzt die Fußgänger-Grün-Phase.
  - Implementierung 2: Die Fußgänger können eine laufende Straßenanmeldung nicht abbrechen.

    Wird eine Fußgängeranmeldung durch eine Straßenanmeldung unterbrochen, so schaltet die Ampel nach einer sehr kurzen Grünphase für die Fußgänger und vorschriftsgemäßen Räumzeiten auf „Straße grün“.
  - Implementierung 4: Die Ampel erzwingt entgegen der Spezifikation Mindestzeiten.

  Die Abweichungen von der Spezifikation müssen explizit benannt werden.

  Außerdem soll bemerkt werden (nicht zwingend durch ein Testfall zu belegen):
  - Implementierung 3: Falls Fußgänger die Straßenanmeldung unterbrechen, herrscht Dauerrot für die Fahrzeuge, da sie schlecht die Induktionsschleife ein zweites Mal überfahren können.
- Teil 2: Die obigen Besonderheiten sollen mit Hilfe geeigneter Testfälle entdeckt werden. Die Testfälle sollen möglichst explizit beschrieben werden. Jeder Testfall muss (sofern möglich) auf alle Implementierungen angewendet werden und das Ergebnis beschrieben werden.
- Teil 3: Mindestens drei sinnvolle Schwächen der Spezifikation sollen erkannt werden, z. B. die folgenden:
  - dass bei ständigem Umschalten der Ampel die nötigen Räumzeiten den Verkehrsfluss stark behindern.
  - das Blockieren allen Verkehrs durch häufiges „Drücken“ von Fußgängern.
  - das „Vergessen“ der Straßenanmeldung bei schnell darauffolgender Fußgängeranmeldung (s.o.).

  Es ist naheliegend, auch auf Lösungsmöglichkeiten – wie die Spezifizierung von Mindestzeiten – einzugehen; gefordert ist das durch die Aufgabenstellung aber nicht.

Aus den Einsendungen: Perlen der Informatik

Allgemeines

Worte des Wettbewerbs: Algorhytmus, Brokolie

… würde ich mich sehr über ein Featback freuen.

Leider habe ich aufgrund akuten Zeitmangels und kurzfristiger Entscheidung am BWINF teilzunehmen diese Aufgaben komplett gelöst.

Einen genaueren Plan, wie das Programm später aussehen würde, hatten wir noch nicht, aber wir machten uns erst einmal an die Arbeit.

Technisches

code-basiertes Programmieren

Killerameisen-Heuristik

Beim Drucken stellte sich heraus, dass mein Drucker nur bei Text ein klares Druckbild erzielt. Bilder wurden leider streifig [..] für die 2. Runde repariere ich das Gerät.

In der größeren for-Schleife findet sich eine kleinere wieder.

Um ein möglichst einfaches Programm zu schreiben, bei dem wahrscheinlich viele Variablen benötigt werden, haben wir die Programmiersprache Javascript benutzt.

Ich habe mein Programm in Javascript geschrieben, da ich hiermit sehr einfach plattformunabhängige GUIs mit einer auf den ersten Anblick sehr mächtigen Sprache erstellen kann.

Die Eingabe wird mit dem Admit Button bestätigt.

Als erstes wird das Programm auf nahezu null zurückgesetzt.

Das Programm wurde in C (← ♡) geschrieben.

Das Programm ist gegen etwaige Eingabenfehler gesichert, lediglich das Schließen des GUI-Fensters ist zu unterlassen.

Einzelne Funktionen sind nach dem Prinzip der objektorientierten Programmierung als seperate und universelle 'functions' aufgebaut … Die Daten … sind hierbei als Datenfelder einer zentralen Klasse 'Core' gespeichert.

Das Programm sollte in einzelnen Schritten arbeiten, damit kein Fehler auftritt und man später mehr Übersicht hat.

Wir haben uns für C++ entschieden, weil es eine übersichtliche Benutzeroberfläche zur Verfügung stellt.

Passendes Wort

Wir kommen in 12 Versuchen!

Und wer lässt schon drei Wochen lang seinen PC laufen, nur um wie Chuck Norris ohne Einladung ins SchülerVZ zu kommen.

… und verlängert entsprechend die knack-Dauer.

Da unsere Welt immer bilingualer wird …

Da dieses Problem keine sehr hohe algorithmische Komplexität hat, kann auf Objektorientierung verzichtet werden.

Pizzavision

Pizzazutaten: Oktopus, Tastatur, Schokoladeneis, Gänseblümchen, Vitamin-C-Pillen, ganze Bananen mit Schale, ganzer Thunfisch, Sushi, Salatgurken, ganze Ananas, Sahne, Nachos, Hühnerkeule

Ich denke mal objektorientiert. Eine Pizza ist eine Pizza.

Die Tatsache, dass unser Programm eine leckere Pizza erstellt und Freude auf mehr Zutaten macht, sehe ich auf Grund einer längeren und freiwilligen(!) Verwendung durch meine Schwester bestätigt.

… man müsste $2^{12} = 4096$ verschiede Pizzas haben oder kaufen.

Wenn der User mit seinen Zusammenstellungen zufrieden ist, sollte er die Pizza so bestellen können. Dies konnte natürlich nicht bis in die letzte Instanz ausgeführt werden, da mir dazu die Pizzabäckerei fehlt!

Ich habe das Pizzaprogramm noch mit CSS verschönert …

```
/**
* Das ist der Grund, warum es keinen Käse gibt
* @ return
*/
public static String getMaus() {
StringBuffer maus = new StringBuffer();
maus.append ("  0___\n");
maus.append (" /.   \\  _\n")
maus.append ("*\\____\\_/\n")
return maus.toString()
}
```

Tankomatik

Spekulanten, Altbundeskanzler und Scheichs haben einen wesentlich größeren Einfluss auf den Ölpreis als die Verbraucher einer Kleinstadt.

Asso sollte einen Tankwart einstellen.

… sollte einmal ein Tankwart mit der nötigen Intelligenz da sein und diese Informationen an die Zentrale weitergeben.

365 Tage, also fast ein Jahr

… außerdem ändert sich nachts der Preis nicht (18-6 Uhr), da die Börsianer pennen.

Wenn es mehr als 60 Kunden in der Stunde gibt, wird überall Fehler angezeigt.

Bitte nicht erschrecken, ich habe das Diagramm aufgrund seiner Größe geteilt.

Durch geringfügige Abweichungen des stündlichen Benzinpreises der Kongruenten …

Bei der nächsten Simulation verwende ich nur ein Zeitintervall von 8760 Tagen, was einem normalen Kalenderjahr entspricht.

Die Prozedur erniedrigt das Unterlimit für Marktpreise um 0.01.

Hierbei nutze ich die Gelegenheit und visualisiere einige Daten in Diagrammen zur optischen Verstärkung.

Der häufigste Fehler im Zusammenhang mit Börsenkursen ist die Annahme, dass die Schwankungen sich durch eine Normalverteilungg beschreiben lassen. Dieser Fehler wird so häufig begangen, dass es nicht schlimm sein kann, wenn ich den gleichen Fehler mache.

Alle Alpen

Der Berg muss fallen!

Keuch, das war anstrengend, erstmal den Augenblick genießen.

… zuerst angweilig geradeaus … und dann noch ein Ausschnitt aus dem VW-Börsenchart.

So erreicht man, dass die Spitzen der Berge nach oben zeigen und das Gebirge nicht an der Decke hängt.

… aber der zur Verfügung stehende Arbeitsspeicher ist deutlich begrenzter als die (theoretische ewig) vorhandene Ausführungszeit.

Aus einem Grund, den ich mittlerweile nicht mehr nachvollziehen kann, habe ich -1 durch 0, 0 durch 1 und 1 durch 2 ausgedrückt.

Hier ist also ein Lösung mittels Brute Force angemessen.

Kreisrund

Die Laufzeit des Algorithmus steigt fakultativ mit der Anzahl der Mitarbeiter an.

Schon nach wenigen Sekunden ist der Algorithmus in großen Suchtiefen verschwunden.

Eltern, die eine bestimmte Teeküche besuchen, heißen User dieser Teeküche.

Ein Teezyklus von 0 würde bedeuten, dass der Kollege seine komplette Arbeitszeit in der Teeküche verbringt, was zur Folge hätte, dass er dort gar nicht mehr arbeiten würde, weil man ihn schon längst rausgeschmissen hätte.

```
Zahl2:= 2000000;
// Dieser Wert indiziert eine irreparable Begrenzungsübertretung
```

Bürgerampel

Vielleicht war Methode 3 aber auch einfach nur ein Modell zur Bekämpfung des Klimawandels, weil sich dabei wohl jeder Autofahrer genau überlegen würde, ob er nicht doch lieber zu Fuß gehen sollte, weil er an der Ampel wahrscheinlich sowieso nicht weiterkommen würde.

Methode 4 der Implementation behebt diese Probleme, indem eine Mindestdauer festgelegt wird, die eine Ampel grün sein muss. In diesem Zeitraum wird eine Bedarfsanmeldung zurückgestellt und später durchgeführt. Dies scheint (nach meinen intensiven Versuchen mit dem lokalen Ampelsystem) auch in Wirklichkeit sichergestellt zu sein.

Da bleibt nur noch eins: Bei Rot fahren.

Ein solches Ampelsystem stellt definitiv zu hohe Ansprüche an die Sozialkompetenz der Betroffenen.