40. Bundeswettbewerb Informatik 2021/2022, 2. Runde

Bundeswettbewerb Informatik

Lösungshinweise

Allgemeines

Es ist immer wieder bewundernswert, wie viel an Ideen, Wissen, Fleiß und Durchhaltevermögen in den Einsendungen zur 2. Runde eines Bundeswettbewerbs Informatik stecken. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir die Arbeiten kritisch begutachten und hohe Anforderungen stellen. Deswegen sind Punktabzüge die Regel und Bewertungen mit Pluspunkten über die Erwartungen hinaus die Ausnahme.

Spannende bzw. schwierige Erweiterungen der Aufgabenstellung sind Extrapunkte wert, wenn sie auch praktisch realisiert wurden. Weitere Ideen ohne Implementierung und geringe Verbesserungen der bereits implementierten Lösung einer Aufgabe gelten allerdings nicht als geeignete Erweiterungen. Intensive theoretische Überlegungen wie z. B. ein korrekter Beweis zur Komplexität des Problems werden ebenfalls mit zusätzlichen Punkten belohnt.

Falls Ihre Einsendung nicht herausragend bewertet wurde, lassen Sie sich auf keinen Fall entmutigen! Allein durch die Arbeit an den Aufgaben und ihren Lösungen hat jede Teilnehmerin und jeder Teilnehmer viel gelernt; diesen Effekt sollten Sie nicht unterschätzen. Selbst wenn Sie nur die Lösung zu einer Aufgabe einreichen konnten, so kann die Bewertung Ihrer Einsendung bei der Anfertigung künftiger Lösungen hilfreich für Sie sein.

Bevor Sie sich in die Lösungshinweise vertiefen, lesen Sie bitte kurz die folgenden Anmerkungen zu den Einsendungen und beiliegenden Unterlagen durch.

Bewertungsbogen

Aus der ersten Runde oder auch aus früheren Wettbewerbsteilnahmen kennen Sie den Bewertungsbogen, der angibt, wie Ihre Einsendung die einzelnen Bewertungskriterien erfüllt hat. Auch in dieser Runde können Sie den Bewertungsbogen im Anmeldesystem AMS einsehen. In der 1. Runde ging die Bewertung noch von 5 Punkten aus, von denen bei Mängeln abgezogen werden konnte. In der 2. Runde geht die Bewertung von zwanzig Punkten aus, bei denen Punkte abgezogen und manchmal auch hinzuaddiert werden konnten. In dieser Runde gibt es auch deutlich mehr Bewertungskriterien als in der 1. Runde.

Terminlage

Für Abiturientinnen und Abiturienten ist der Terminkonflikt zwischen Abiturvorbereitung und 2. Runde sicher nicht ideal. Doch leider bleibt dem Bundeswettbewerb Informatik nur die erste Jahreshälfte für diese Runde: In der zweiten Jahreshälfte läuft nämlich die zweite Runde des

Mathematikwettbewerbs, dem wir keine Konkurrenz machen wollen. Aber: die Bearbeitungszeit für die Runde beträgt etwa vier Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der wichtigen, ausführlichen Dokumentation der Aufgabenlösungen zu vermeiden. Aufgaben der 2. Runde sind oft deutlich schwerer zu lösen, als sie auf den ersten Blick erscheinen. Erst bei der konkreten Umsetzung einer Lösungsidee stößt man manchmal auf Besonderheiten bzw. noch zu lösende Schwierigkeiten, was dann zusätzlicher Zeit bedarf. Daher ist es sinnvoll, die einzureichenden Aufgaben nicht nacheinander, sondern relativ gleichzeitig zu bearbeiten, um nicht vom zeitlichen Aufwand der jeweiligen Aufgabe unangenehm kurz vor Ablauf der Bearbeitungszeit überrascht zu werden.

Dokumentation

Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und deren Umsetzung in Software fließen lassen. Doch ohne eine verständliche Beschreibung der Lösungsideen und ihrer jeweiligen Umsetzung, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine geeignete Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (welche die verschiedenen, bei der Lösung des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung nur wenig wert.

Bewerterinnen und Bewerter können die Qualität Ihrer Aufgabenlösungen nur anhand dieser Informationen vernünftig einschätzen. Mängel in der Dokumentation der Einsendung können nur selten durch Ausprobieren und Testen der Programme ausgeglichen werden – wenn die Programme denn überhaupt ausgeführt werden können: Hier gibt es gelegentlich Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner auf Lauffähigkeit getestet würden. Insgesamt sollte die Erstellung der Dokumentation die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt auch keine saubere Umsetzung hin, in welcher Programmiersprache auch immer.

Einige unterhaltsame Formulierungsperlen sind im Anhang wiedergegeben.

Bewertung

Bei den im Folgenden beschriebenen Lösungsideen handelt es sich nicht um perfekte Musterlösungen, sondern um sinnvolle Lösungsvorschläge. Dies sind also nicht die einzigen Lösungswege, die wir gelten ließen. Wir akzeptieren vielmehr in der Regel alle Ansätze, auch ungewöhnliche, kreative Bearbeitungen, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige der Fußnoten in den folgenden Lösungsvorschlägen verweisen auf weiterführende Fachliteratur für besonders Interessierte; Lektüre und Verständnis solcher Literatur wurden von den Teilnehmenden natürlich nicht erwartet.

Unabhängig vom gewählten Lösungsweg gibt es aber Dinge, die auf jeden Fall von einer guten Lösung erwartet wurden. Zu jeder Aufgabe wird deshalb in einem eigenen Abschnitt, jeweils am Ende des Lösungsvorschlags erläutert, auf welche Kriterien bei der Bewertung dieser Aufgabe besonders geachtet wurde. Dabei können durchaus Anforderungen formuliert werden, die aus der Aufgabenstellung nicht hervorgingen. Letztlich dienen die Bewertungskriterien dazu,

die allerbesten unter den sehr vielen guten Einsendungen herauszufinden. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation (klare Beschreibung der Lösungsidee, genügend aussagekräftige Beispiele und wesentliche Auszüge aus dem Quellcode) einschließlich einer theoretischen Analyse (geeignete Laufzeitüberlegungen bzw. eine Diskussion der Komplexität des Problems) sowie an den Quellcode der implementierten Software (mit übersichtlicher Programmstruktur und verständlicher Kommentierung) und an das lauffähige Programm (ohne Implementierungsfehler). Wünschenswert sind auch Hinweise auf die Grenzen des angewandten Verfahrens sowie sinnvolle Begründungen z. B. für Heuristiken, vorgenommene Vereinfachungen und Näherungen. Geeignete Abbildungen und eigene zusätzliche Eingaben können die Erläuterungen in der Dokumentation gut unterstützen. Die erhaltenen Ergebnisse für die Beispieleingaben (ggf. mit Angaben zur Rechenzeit) sollten leicht nachvollziehbar dargestellt sein, z. B. durch die Ausgabe von Zwischenschritten oder geeignete Visualisierungen. Eine Untersuchung der Skalierbarkeit des eingesetzten Algorithmus hinsichtlich des Umfangs der Eingabedaten ist oft ebenfalls nützlich.

Danksagung

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

An der Erstellung der im folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Julian Baldus (Aufgabe 1), Simon Schwarz (Aufgabe 1), Boldizsár Mann (Aufgabe 2), Manuel Gundlach (Aufgabe 3) und Hans-Martin Bartram (Bonusaufgabe). Allen Beteiligten sei für ihre Mitarbeit ganz herzlich gedankt.

Aufgabe: Müllabfuhr

1.1 Lösungsidee

Wir betrachten das Straßennetz als Graphen. Sei $G = (V, E)$ der Graph, der das Straßennetz darstellt. Dabei ist V die Menge der Knoten (Kreuzungen) und E die Menge der Kanten (Straßen).

Das gestellte Problem ist sehr schwer. Genauer gesagt ist es NP-schwer, selbst wenn man nur zwei statt fünf Tage betrachtet.[1] Gleichzeitig enthalten die vorgegebenen Beispieleingaben bis zu 1000 Knoten. Wir können also nicht erwarten, eine optimale Lösung zu finden. Stattdessen setzen wir auf Heuristiken.

Dazu werden wir in zwei Schritten vorgehen:

1. Finde einen Weg, der am Startknoten anfängt und aufhört und der alle Kanten mindestens einmal besucht.
2. Teile den gefundenen Weg in fünf möglichst gleich große Teile auf, wobei Anfang und Ende jedes Abschnitts mit dem Start verbunden werden müssen.

Diese beiden Teile können wir optimal lösen. Es ist jedoch wichtig zu erkennen, dass die Gesamtlösung trotzdem nicht immer optimal ist: Eventuell gäbe es einen etwas längeren (oder einen ebenso langen) Gesamtweg, der sich aber besser in fünf Teile teilen lässt und dadurch eine bessere Lösung erzielen würde.

Finden eines Weges, der alle Kanten abdeckt

Die Aufgabe, einen möglichst kurzen Weg zu finden, der alle Kanten abdeckt, ist ähnlich zu dem Eulerkreisproblem. Dabei wird ein Weg gesucht, der jede Kante *genau* einmal besucht.

Sei der *Grad eines Knotens v* die Anzahl Kanten, die an v angrenzen. Ein Eulerkreis existiert genau dann, wenn jeder Knoten im Graphen einen geraden Grad hat. Das lässt sich dadurch begründen, dass jedes Mal, wenn der Weg einen Knoten besucht, eine angrenzende Kante *verbraucht* wird, und man eine weitere Kante braucht, um den Knoten wieder zu verlassen. Wenn ein Knoten einen ungeraden Grad hat, würde man unweigerlich an diesem Knoten *stecken bleiben*. Ein Eulerkreis lässt sich mit dem Algorithmus von Hierholzer effizient finden, wenn er existiert. Dabei geht man zuerst einen beliebigen Pfad durch den Graphen, bis man wieder beim Ausgangsknoten ankommt. Es ist garantiert, dass man wieder beim Ausgangsknoten ankommt, da alle Knoten einen geraden Grad haben. Dann betrachtet man einen Knoten, bei dem noch angrenzende Kanten unbesucht sind und sucht einen weiteren Pfad von dort aus, und fügt diesen in den bisherigen Kreis ein. Dies wird so lange wiederholt, bis alle Kanten besucht wurden.

Falls in dem gegebenen Graphen also nun alle Knoten einen geraden Grad haben, können wir durch Bestimmung des Eulerkreises verhältnismäßig einfach den kürzesten Weg finden, der alle Kanten abdeckt. Was machen wir nun, wenn ein solcher Eulerkreis nicht existiert?

[1] Wenn man das gestellte Problem effizient lösen könnte, dann könnte man auch eine Variante von Subset Sum effizient lösen, indem man für jedes Element der gegebenen Zahlenmenge eine Kante vom Startknoten zum Startknoten mit dieser Länge hinzufügt. Eine effiziente Lösung des Müllabfuhrproblems fände nun eine Aufteilung der Kanten in zwei Teilmengen mit gleich großer Summe, wenn vorhanden. Es ist aber bekannt, dass dies NP-schwer ist, daher muss auch das Müllabfuhrproblem NP-schwer sein.

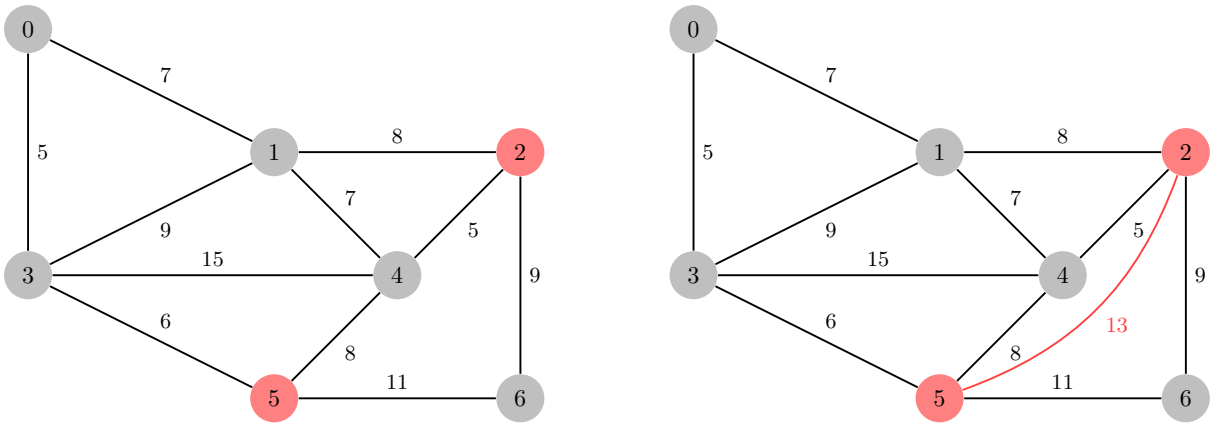

Abbildung 1.1: Beispiel für einen Straßenplan, wo jede Kreuzung durch einen Knoten und jede Straße als eine Kante repräsentiert ist. Die Länge der Straße ist das Gewicht der Kante. Knoten mit geradem Grad sind grau und Knoten mit ungeradem Grad rot markiert. Rechts sind die beiden roten Knoten mit einer Metakante verbunden, welche als Gewicht die kürzeste Verbindung zwischen den beiden Knoten hat. In dem rechten Straßenplan gibt es damit einen Eulerzyklus.

In dem Fall werden wir einige Kanten mehrfach besuchen müssen. Unsere Aufgabe ist es, die Gesamtlänge der mehrfach besuchten Kanten zu minimieren.

Betrachten wir einmal alle Knoten in dem gegebenen Graphen mit ungeradem Grad. Wenn wir jeweils zwei dieser Knoten mit einer neuen Kante verbinden, dann hätte der resultierende Graph einen Eulerzyklus. Praktischerweise wird es auch immer eine gerade Anzahl solcher Knoten geben, da jede Kante zwei Enden hat und daher die Gesamtzahl der Start- und Endpunkte der Kanten gerade ist. Wir fügen nun also *Metakanten* in den Graphen ein. Diese haben jeweils die Länge des kürzesten Weges zwischen zwei Knoten. Wir verbinden je zwei Knoten mit ungeradem Grad mit einer Metakante, sodass nachher jeder Knoten einen geraden Grad hat. Auf dem entstehenden Graphen können wir nun einen Eulerzyklus finden. Dieser deckt alle ursprünglichen Kanten ab. Die Länge entspricht der Länge aller ursprünglichen Kanten plus der Länge der eingefügten Kanten. Wir müssen also nur die Gesamtlänge der eingefügten Kanten minimieren.[2] Das Vorgehen ist exemplarisch in Abbildung 1.1 dargestellt.

Wir müssen nun also alle Knoten mit ungeradem Grad so in Paare aufteilen, dass die Summe der kürzesten Wege zwischen den Partnern minimal ist. Die kürzesten Wege zwischen allen relevanten Knoten können wir zum Beispiel mit dem Dijkstra-Algorithmus oder auch mit dem Algorithmus von Floyd und Warshall finden. Die Aufteilung in Paare ist bekannt als *Matching*. Konkret handelt es sich um *Min Cost Perfect Matching*. Im Gegensatz zu Matching auf bipartiten Graphen ist Matching auf allgemeinen Graphen leider umständlich. Trotzdem gibt es mit dem Blossom-Algorithmus auch für dieses Problem eine bekannte Lösung. Falls man diesen jedoch nicht implementieren möchte, kann man auch schon gute Ergebnisse erreichen, indem man die Knotenpaare mit einem gierigen Algorithmus zuweist – also zum Beispiel über alle Knoten iteriert, und jedem nicht zugewiesenen Knoten den ebenfalls nicht zugewiesenen Knoten mit der kleinsten Distanz zuweist.

[2] Es ist übrigens auch garantiert, dass wir einen Weg minimaler Länge mit dieser Methode finden können, also in mindestens einem optimalen Weg alle mehrfach besuchten Kanten zu solchen *Metakanten* zusammengefügt werden können. Den Beweis bleiben wir hier jedoch schuldig.

Aufteilung des Weges in Tagesabschnitte

Nun haben wir also einen Weg, der alle Kanten abdeckt und so kurz wie möglich ist. Die nächste Aufgabe ist, diesen in fünf Teile aufzuteilen, sodass die Länge des längsten Teils minimal ist. Komplikationen dabei sind:

- Der Weg kann jeweils nur an den Knoten, nicht in der Mitte einer Kante geteilt werden.
- Die Enden jedes Abschnittes müssen mit dem Startknoten verbunden werden.

Betrachten wir zunächst einmal eine etwas einfachere Aufgabenstellung: Gegeben eine Länge x, können wir den Zyklus in 5 Tagesabschnitte der Länge maximal x aufteilen? Um das zu beantworten, starten wir beim Startknoten und wählen von dort den längsten Tagesabschnitt, dessen Länge höchstens x ist. Für den nächsten Tag starten wir bei der ersten Kante, die noch nicht verwendet wurde, und wählen von dort wieder den längsten Abschnitt mit Länge höchstens x, und führen das für die übrigen Tage fort. Wenn wir nach fünf Tagen alle Kanten abgedeckt haben, ist die Länge x ausreichend.

Wir können nun eine binäre Suche ausführen, um das kleinste x zu finden, für das wir den Zyklus noch in entsprechend lange Tagesabschnitte teilen können. Wir starten mit einem Minimalwert, für den bekannt ist, dass er als Länge der Abschnitte nicht ausreicht (zum Beispiel 0) und einem Maximalwert, der auf jeden Fall ausreichen würde (zum Beispiel die Gesamtlänge des Zyklus). Dann prüfen wir für den Mittelwert zwischen Minimalwert und Maximalwert, ob er ausreichend ist. Wenn ja, ist der gesuchte Wert höchstens so groß wie der Mittelwert, ansonsten ist er größer. Dementsprechend werden die Maximal- und Minimalwerte angepasst und das Verfahren wiederholt. So finden wir sehr schnell das gesuchte x. Dieses wird die Länge des längsten Tagesabschnitts in unserer Lösung sein.

Laufzeit

Um die Laufzeit des Algorithmus betrachten zu können, definieren wir erst einmal $N = |V|$ und $M = |E|$, i. e. N ist die Anzahl Knoten in dem vorgegeben Graphen und M ist die Anzahl Kanten. Sei außerdem L die Gesamtlänge aller Kanten in der Eingabe.

Zuerst findet der Algorithmus Paare von Knoten mit ungeradem Grad. Dazu müssen wir zuerst die kürzesten Wege zwischen allen Paaren solcher Knoten finden. Der Algorithmus von Floyd und Warshall braucht hierfür eine Laufzeit in $\mathcal{O}(N^3)$. Mit Anwendung des Dijkstra-Algorithmus für jeden Knoten erhalten wir stattdessen eine Laufzeit von $\mathcal{O}(N(N+M)\log N)$, was besser ist, sofern M deutlich kleiner als N^2 ist.

Das Zuteilen der Paare selbst kann dann durchaus in linearer Laufzeit ($\mathcal{O}(N+M)$) geschehen, falls hier ein Greedy-Algorithmus gewählt wird. Falls man jedoch den Blossom-Algorithmus von Edmonds verwendet, um optimale Ergebnisse zu erhalten, wird es länger dauern. Der Blossom-Algorithmus kann mit einer Laufzeit von $\mathcal{O}(n^2m)$ für n Knoten und m Kanten umgesetzt werden. Relevant sind hier jedoch nicht die Kanten in dem Ausgangsgraphen, sondern die Knoten mit ungeradem Grad (bis zu N) und alle $\mathcal{O}(N^2)$ kürzesten Wege zwischen ihnen als Kanten. Die Laufzeit ist also bis zu $\mathcal{O}(N^4)$. Jedoch haben häufig deutlich weniger Knoten einen ungeraden Grad. In den Beispielen vom BWINF sind es nur maximal 240. Außerdem wird die angegebene Worst-Case-Laufzeit des Blossom-Algorithmus nur selten erreicht. Daher kann dieser Schritt auf allen vorgegebenen Beispielen in unter einer Sekunde durchlaufen, wenn

eine effiziente Implementierung dieses Algorithmus verwendet wird.[3].

Danach muss auf dem Graphen ein Eulerkreis gefunden werden. Der Algorithmus von Hierholzer benötigt hierfür nur lineare Laufzeit, also $\mathcal{O}(N+M)$.

Zum Aufteilen des Pfades werden logarithmisch viele Iterationen der binären Suche benötigt, das sich die Größe des betrachteten Intervalls in jedem Schritt halbiert. Jede Iteration betrachtet dabei den gesamten Zyklus der Länge $\mathcal{O}(M)$. Dieser Schritt hat also eine Laufzeit von $\mathcal{O}(M \log L)$.

Die Gesamtlaufzeit ist also (bei Verwendung des Floyd-Warshall- und des Blossom-Algorithmus) $\mathcal{O}(N^3+N^4+(N+M)+M\log L)$. Dies lässt sich grob zu $\mathcal{O}(N^4)$ vereinfachen, da $M < N^2$ und auch L zumindest in den vorgegebenen Eingaben nicht übermäßig groß ist. Auch in der Praxis stellt sich heraus, dass der Blossom-Algorithmus die Laufzeit dominiert.

Weitere Verbesserungen

Da es sich bei dem vorgestellten Algorithmus nur um eine Heuristik handelt, sind die Ergebnisse nicht optimal. Bei einer guten Implementierung liegt die Laufzeit außerdem erst bei ca. einer Sekunde auf den größten vorgegebenen Beispielen. Daher kann man sich Gedanken machen, wie die Ergebnisse noch weiter verbessert werden können. Ein paar Ideen sind hier grob skizziert:

- Ein Graph, der einen Eulerzyklus besitzt, hat typischerweise nicht nur einen, sondern viele verschiedene Eulerzyklen. Die Auswahl des konkreten Eulerzyklus kann außerdem einen Einfluss auf die Qualität des Ergebnisses haben. Es ist jedoch nicht direkt klar, welcher Eulerzyklus der beste ist. Bisher finden wir einen beliebigen. Da jedoch die Laufzeit für das Finden des Eulerzyklus und die darauf folgende Aufteilung des Zyklus bisher nur einen kleinen Teil der Gesamtlaufzeit ausmacht, können wir auch einfach mehrere, z. B. 100 verschiedene Eulerzyklen ausprobieren, um mit gewisser Wahrscheinlichkeit ein etwas besseres Ergebnis zu erreichen. Verschiedene Eulerzyklen können zum Beispiel durch zufälliges durchmischen der Kanten jedes Knotens vor der Eulerzyklussuche gefunden werden.
- Der Eulerzyklus, der gefunden wurde, enthält einige Kanten, die wir nachträglich eingefügt haben, um einen Eulerzyklus zu ermöglichen. Diese Kanten müssen nicht unbedingt befahren werden. Falls die erste oder letzte Kante eines Tagesabschnitts eine solche Kante ist, können wir sie entfernen und den Tagesabschnitt dadurch kürzer machen. Für möglichst gute Ergebnisse sollte dies auch bereits bei der Aufteilung in Tagesabschnitte berücksichtigt werden.
- Bisher gehen wir davon aus, dass unser Eulerzyklus einen definierten Start- beziehungsweise Endpunkt hat, und dass der erste Tagesabschnitt auch an diesem Punkt startet. Es kann sich jedoch lohnen, zu erlauben, dass ein Tagesabschnitt diesen Startpunkt *überlappt*. Dies lässt sich erreichen, ohne die Laufzeit wesentlich zu verschlechtern: Falls die Aufteilung des Zyklus in Tagesabschnitte in einer Iteration der binären Suche nicht geklappt hat, versucht man, den ersten Abschnitt um eine Kante nach vorne zu *verschieben* und passt auch die Enden der übrigen Abschnitte entsprechend an. Falls dadurch der letzte Abschnitt lang genug werden kann, um auch die ersten, nun frei gewordenen Kanten

[3]Getestet haben wir das mit der Implementierung „Blossom V“, die hier beschrieben wird: `https://pub.ist.ac.at/~vnk/papers/blossom5.pdf`

des Zyklus abzudecken, ist man fertig. Ansonsten probiert man, das noch so lange zu wiederholen, bis der erste Abschnitt mindestens die ursprüngliche Position des zweiten Abschnitts erreicht hat.

Qualität der Ergebnisse

Da wir – wie oben festgestellt – mit einer Heuristik *keine optimalen* Ergebnisse erreichen, kann analysiert werden, wie gut die berechneten Ergebnisse tatsächlich sind. Dazu wollen wir eine *untere Schranke* verwenden, um abzuschätzen, welche Länge ein optimales Ergebnis immer haben muss.

Zuerst kann man feststellen, dass die gefundene Länge des längsten Tagesabschnittes maximal fünf mal so lang ist wie der längste Tagesabschnitt in der optimalen Lösung. Wenn wir nämlich alle Tagesabschnitte aus der optimalen Lösung zusammenfügen, hätten wir eine Lösung für nur einen Tag, und diese können wir mit unserem Algorithmus optimal berechnen. Durch das Aufteilen dieses Weges in fünf Tagesabschnitte wird die Länge nur kleiner.

Wir können jedoch noch genauere Aussagen treffen: Wenn man eine optimale Lösung hätte und bei dieser alle Tagestouren zusammenfügen würde, erhält man einen Zyklus, der alle Kanten abdeckt und einige Kanten möglicherweise mehrfach abdeckt. Die Länge der längsten Tagestour ist zumindest ein Fünftel der Gesamtlänge dieses Zyklus. Der Eulerzyklus auf dem erweiterten Graphen, den wir berechnen, ist ebenfalls ein solcher Zyklus auf dem Originalgraphen. Und wir wissen, dass unser Eulerzyklus minimale Länge hat (sofern wir den Blossom-Algorithmus für das Matching verwenden). Daher hat auch der Zyklus aus der optimalen Lösung mindestens diese Länge. Teilen wir nun also die Länge unseres Zyklus durch fünf, erhalten wir eine untere Schranke für die Länge der längsten Tagestour in der optimalen Lösung.

Die untere Schranke erlaubt es, unsere Lösungen einzuordnen. Sollte beispielsweise unsere Lösung sehr nah an der unteren Schranke sein, wissen wir, dass hier sicher nur noch wenig Optimierungsbedarf besteht. Aber Achtung: In die andere Richtung lässt sich keine Aussage treffen. Nur weil unsere Lösung weit von der unteren Schranke entfernt ist, lässt sich nicht direkt sagen, dass wir noch viel optimieren können. Es könnte schließlich auch sein, dass die untere Schranke weit von der optimalen Lösung entfernt ist. Jedoch gilt immer: Lösungen, welche kürzere Tagesstrecken als die angegebene untere Schranke berechnen, müssen falsch sein.

1.2 Alternative Lösung: ILP

Unser letzter vorgestellter Algorithmus benutzte eine *Heuristik*, um gute Ergebnisse zu finden. Allerdings sind diese Ergebnisse natürlich nicht in allen Fällen optimal. In diesem Abschnitt wird kurz eine Möglichkeit vorgestellt, welche optimale Ergebnisse auf allen Instanzen liefert. Allerdings: Da unser Problem NP-schwer ist, wird die folgende Möglichkeit also eine hohe Laufzeit haben – sogar so hoch, dass mit dieser Methode die großen Beispiele des BWINF nicht gelöst werden können. Der folgende Lösungsansatz komplementiert also die Heuristik und kann beispielsweise auf kleinen Instanzen eingesetzt werden, um dort garantiert das beste Ergebnis zu erreichen.

Der Lösungsansatz ergibt sich mit einer Reduktion auf das „Integer Linear Programming"-Problem (kurz *ILP* genannt). Dieses Problem besteht aus eine Menge an *ganzzahligen* Variablen, welche durch *Ungleichungen* eingeschränkt werden. Eine Lösung für das ILP-Problem

besteht aus einer Belegung der Variablen, mit welcher alle Ungleichungen erfüllt sind. Weiterhin kann eine Kostenfunktion angegeben werden. Mit dieser ist es möglich, den Wert einer Variablen zu minimieren. Eine ganzzahlige Lösung, welche alle Ungleichungen erfüllt sowie die Kosten minimiert, nennen wir *optimal*.

Beispielsweise könnte ein ILP-Problem wie folgt aussehen:

$$\begin{array}{lll} \text{minimiere} & x & \\ \text{gegeben} & x+y & \geq 4 \\ & 2x-3y & \geq 19 \\ & x,y \in \mathbb{Z} & \end{array}$$

Eine optimale Lösung für dieses Problem wäre nun $x = 7$, $y = -2$.

ILP ist ebenfalls NP-schwer[4]. Daher gibt es ebenfalls keinen effizienten Algorithmus, um eine optimale Lösung für eine ILP-Instanz zu finden. Jedoch wird sehr intensiv an Algorithmen geforscht, welche zwar theoretisch eine hohe Laufzeit haben, in der Praxis aber doch sehr viele Instanzen schnell lösen können. Solche Programme, wie z. B. GLPK[5], werden „ILP-Solver“ genannt.

Unsere Reduktion

Unser Ziel ist es, eine Instanz des Müllabfuhr-Problems mit der Hilfe eines ILP-Solvers zu lösen. Dazu müssen wir für eine gegebene Müllabfuhr-Instanz eine Instanz des ILP-Problems konstruieren, dieses lösen, und schlussendlich aus der Lösung des ILP-Problems unseren Tagesplan rekonstruieren. Diese Vorgehensweise heißt in der Informatik eine *Reduktion*: Wir *reduzieren* das Müllabfuhr-Problem auf ILP.

Dies bedeutet, dass wir für einen Müllabfuhr-Eingabegraph $G = (V, E)$ eine Menge an Ungleichungen definieren müssen, sodass diese einen Tagesplan ergeben. Zu diesem Zweck führt unsere Reduktion für alle Kanten in $(i, j) \in E$ fünf Variablen x_{ij1} bis x_{ij5} ein. Jede dieser Variablen soll später für die Anzahl stehen, wie oft die Kante an einem bestimmten Tag durchfahren werden soll. Nun können wir mit Hilfe dieser Variablen unser Ziel beschreiben. Beispielsweise soll jede Kante an mindestens einem Tag durchfahren werden. Dies ergibt die folgenden Ungleichungen:

$$\sum_{k=1}^{5} x_{ijk} \geq 1 \quad \forall (i, j) \in E$$

Natürlich müssen noch deutlich mehr Ungleichungen hinzugefügt werden. So muss natürlich sichergestellt werden, dass jede Tour an der Zentrale beginnt und endet. Ebenfalls dürfen natürlich nur Kanten befahren werden, wenn wir diese auf dem Weg erreichen können. Aus Platzgründen werden hier nicht alle Ungleichungen explizit beschrieben. Diese können jedoch zum Beispiel in der Arbeit „On the balanced K-chinese Postmen Problems“[6] nachgelesen werden.

Weiterhin kann mit diesem Ansatz eine Kostenfunktion definiert werden. Dazu wählen wir die maximale Tageslänge aus, welche minimiert werden soll. Insgesamt erhalten wir somit eine Menge an Ungleichungen und eine Kostenfunktion. Für dieses Problem können wir nun einen ILP-Solver befragen, welcher uns die Belegung der Variablen für eine optimale Lösung ausgibt.

[4] Sonst könnten wir unser originales Problem nicht effizient auf ILP reduzieren.

[5] `https://www.gnu.org/software/glpk/`

[6] `https://etd.lib.metu.edu.tr/upload/12618933/index.pdf`, Seiten 13 – 14

Da die Variablen die Informationen enthalten, an welchen Tag welche Straße befahren werden soll, können wir nun sehr einfach einen Tagesplan erstellen[7].

Untere Schranken

Da das ILP-Problem NP-schwer ist, kann unsere Lösung auf den großen gegebenen Beispielen (muellabfuhr5.txt bis muellabfuhr8.txt) in vernünftiger Zeit keine Lösung finden. Trotzdem können wir aus unserem Lösungsansatz Infos über die optimale Lösung finden. Dazu benutzen wir die folgende Beobachtung:

> Wenn wir eine optimale Lösung mit *rationalen Zahlen* für eine ILP-Instanz finden, dann ist jede Lösung mit *ganzen Zahlen* schlechter (oder gleich) zu der rationalen Lösung.

Dieser Satz gilt, da alle Lösungen mit nur ganzen Zahlen auch direkt Lösungen mit rationalen Zahlen sind. Diese Beobachtung ist sehr nützlich, da es möglich ist, rationale Lösungen für ILP-Probleme *effizient* zu berechnen. In der Informatik nennt man diese Methode LP-Relaxation.

Mithilfe der LP-Relaxation können wir zwar keine optimale Lösung finden, jedoch erneut eine *untere Schranke* für alle Lösungen. Da jeder Tagesplan auch immer alle Ungleichungen erfüllen muss, kann ein Tagesplan nicht besser als die optimale (rationale) Lösung zu unseren Ungleichungen sein. Allerdings kann es durchaus sein, dass die untere Schranke ein deutlich kleinerer Wert als die optimale Lösung ist; insbesondere muss keine Lösung mit Kosten gleich der unteren Schranke existieren.

1.3 Einfachere Ansätze

Beide vorgestellten Ansätze benötigen viel algorithmisches Vorwissen. Es sind auch andere Ansätze denkbar.

Ein einfacherer heuristischer Ansatz wäre zum Beispiel, vom Startpunkt aus fünf Wege zu suchen, wobei immer an den bisher kürzesten Weg die am schnellsten erreichbare, noch nicht verwendete Kante angefügt wird.

Eine Alternative für diesen Ansatz wäre auch, die Wege nacheinander zu erstellen. Dazu müsste man zuerst eine Maximallänge festlegen, und dann nacheinander fünf Pfade berechnen, die möglichst viele zuvor noch nicht besuchte Kanten enthalten, und dabei diese Maximallänge nicht überschreiten. Bei der Auswahl der Kanten wird auch immer die am schnellsten erreichbare verwendet. Falls die Maximallänge zu gering gewählt ist, werden so nicht alle Kanten abgedeckt. Falls man es aber doch schafft, alle Kanten abzudecken, hat man eine Lösung mit dieser Maximallänge als längste Länge einer Tagestour. Man kann nun binäre Suche anwenden, um eine kleine Maximallänge zu finden, bei der noch eine Lösung gefunden wird.

Genau genommen stimmt es bei diesem Ansatz nicht ganz, dass, wenn eine Lösung für eine Maximallänge gefunden wird, auch für alle größeren Maximallängen eine Lösung gefunden werden wird. Daher kann es sich lohnen, in der binären Suche die obere beziehungsweise untere Grenze dem geprüften Wert nur anzunähern, statt sie direkt auf diesen Wert zu setzen.

[7] Wir suchen für jeden Tag einen Eulerkreis auf den für diesen Tag ausgewählten Kanten, dieser wird unsere Tour.

1.4 Ergebnisse auf den Beispieleingaben

Die folgende Tabelle fasst die Ergebnisse der verschiedenen Ansätze zusammen. In den Spalten ist jeweils die *Länge der längsten Tagesstrecke*, welche von der jeweiligen Lösung berechnet wurde, eingetragen.

Die betrachteten Ansätze sind die folgenden:

ILP Optimale Lösung

H1 Zu Beginn beschriebene Heuristik, die den Blossom-Algorithmus zum Zuteilen von Knotenpaaren verwendet.

H2 Wie H1, jedoch wurden die Knotenpaare mit einem Greedy-Algorithmus zugeteilt.

G1 Greedy-Algorithmus, der parallel fünf Pfade erstellt.

G2 Greedy-Algorithmus, der mit binärer Suche eine Maximallänge bestimmt und für diese nacheinander fünf Pfade erstellt.

G3 Wie G2, aber mit modifizierter binäre Suche, die sich dem geprüften Wert langsamer annähert.

Eingabedatei	ILP	H1	H2	G1	G2	G3	Untere Schranke
`muellabfuhr0`	4	4	4	4	4	4	4
`muellabfuhr1`	18	18	18	25	18	18	18
`muellabfuhr2`	9	10	10	14	10	10	9
`muellabfuhr3`	21	22	22	24	22	22	21
`muellabfuhr4`	10	10	10	10	10	10	10
`muellabfuhr5`	n/a	1468	1468	1479	1468	1468	1464
`muellabfuhr6`	n/a	525291	558100	647578	631102	618121	477419
`muellabfuhr7`	n/a	794793	799288	1236244	813009	813009	447294
`muellabfuhr8`	n/a	2719313	2719313	3355168	3123746	3122164	2666041

Im Folgenden werden die Ergebnispfade für einige Dateien aufgeführt. Aus Platzgründen werden die konkreten Wege jedoch nur für die Dateien bis `muellabfuhr4.txt` angegeben.

muellabfuhr0.txt

Diese Lösung war bereits in der Aufgabenstellung vorgegeben.

```
Tag 1: 0 -> 8 -> 9 -> 8 -> 0, Gesamtlaenge: 4
Tag 2: 0 -> 4 -> 3 -> 2 -> 0, Gesamtlaenge: 4
Tag 3: 0 -> 8 -> 7 -> 6 -> 0, Gesamtlaenge: 4
Tag 4: 0 -> 8 -> 1 -> 2 -> 0, Gesamtlaenge: 4
Tag 5: 0 -> 6 -> 5 -> 4 -> 0, Gesamtlaenge: 4
Maximale Laenge einer Tagestour: 4
```

muellabfuhr1.txt

```
Tag 1: 0 -> 6 -> 3 -> 5 -> 0, Gesamtlaenge: 11
Tag 2: 0 -> 6 -> 3 -> 2 -> 3 -> 6 -> 0, Gesamtlaenge: 18
Tag 3: 0 -> 6 -> 7 -> 5 -> 4 -> 3 -> 6 -> 0, Gesamtlaenge: 18
Tag 4: 0 -> 6 -> 7 -> 6 -> 3 -> 1 -> 6 -> 0, Gesamtlaenge: 15
Tag 5: 0 -> 6 -> 7 -> 4 -> 0, Gesamtlaenge: 16
Maximale Laenge einer Tagestour: 18
```

muellabfuhr2.txt

```
Tag 1: 0 -> 5 -> 11 -> 8 -> 12 -> 8 -> 7 -> 9 -> 5 -> 0, Gesamtlaenge: 9
Tag 2: 0 -> 9 -> 12 -> 1 -> 7 -> 11 -> 2 -> 10 -> 9 -> 0, Gesamtlaenge: 9
Tag 3: 0 -> 6 -> 4 -> 3 -> 11 -> 3 -> 13 -> 1 -> 6 -> 0, Gesamtlaenge: 9
Tag 4: 0 -> 9 -> 7 -> 14 -> 8 -> 2 -> 14 -> 6 -> 9 -> 0, Gesamtlaenge: 9
Tag 5: 0 -> 9 -> 13 -> 14 -> 13 -> 4 -> 10 -> 14 -> 5 -> 0, Gesamtlaenge: 9
Maximale Laenge einer Tagestour: 9
```

muellabfuhr3.txt

```
Tag 1: 0 -> 14 -> 2 -> 1 -> 12 -> 14 -> 11 -> 13 -> 10 -> 2 -> 11 -> 1 -> 10
  -> 11 -> 12 -> 0 -> 10 -> 9 -> 0 -> 11 -> 6 -> 0, Gesamtlaenge: 21
Tag 2: 0 -> 2 -> 12 -> 13 -> 14 -> 7 -> 11 -> 8 -> 5 -> 14 -> 6 -> 13 -> 5
  -> 2 -> 6 -> 10 -> 7 -> 4 -> 2 -> 13 -> 1 -> 0, Gesamtlaenge: 21
Tag 3: 0 -> 13 -> 4 -> 14 -> 10 -> 5 -> 11 -> 9 -> 7 -> 5 -> 9 -> 1 -> 14
  -> 3 -> 11 -> 4 -> 5 -> 1 -> 4 -> 10 -> 3 -> 0, Gesamtlaenge: 21
Tag 4: 0 -> 7 -> 13 -> 9 -> 12 -> 10 -> 8 -> 13 -> 3 -> 12 -> 5 -> 6 -> 12
  -> 8 -> 4 -> 12 -> 7 -> 1 -> 6 -> 3 -> 4 -> 0, Gesamtlaenge: 21
Tag 5: 0 -> 8 -> 14 -> 9 -> 6 -> 4 -> 9 -> 2 -> 7 -> 3 -> 9 -> 8 -> 7 -> 6
  -> 8 -> 2 -> 3 -> 1 -> 8 -> 3 -> 5 -> 0, Gesamtlaenge: 21
Maximale Laenge einer Tagestour: 21
```

muellabfuhr4.txt

```
Tag 1: 0 -> 9 -> 8 -> 9 -> 0 -> 1 -> 2 -> 1 -> 0, Gesamtlaenge: 8
Tag 2: 0 -> 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0, Gesamtlaenge: 10
Tag 3: 0 -> 9 -> 8 -> 9 -> 0, Gesamtlaenge: 4
Tag 4: 0 -> 1 -> 0, Gesamtlaenge: 2
Tag 5: 0 -> 1 -> 0, Gesamtlaenge: 2
Maximale Laenge einer Tagestour: 10
```

1.5 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Das Straßennetz kann sinnvoll als Graph dargestellt werden, dessen Kanten z. B. in einer Adjazenzliste gespeichert werden.

(2) *Laufzeit des Verfahrens in Ordnung*: Alle vorgegebenen Testfälle sollten in wenigen Minuten bearbeitet werden können. Falls ein optimales Verfahren gewählt wurde, reicht es, wenn damit alle Testfälle bis einschließlich `muellabfuhr4.txt` in angemessener Zeit bearbeitet werden können. Solch ein Verfahren muss aber mit einer Heuristik ergänzt werden, sodass auch auf größeren Eingaben Ergebnisse geliefert werden.

(3) *Speicherbedarf in Ordnung*: Das Programm sollte nicht unnötig viel (mehr als einige GB) Speicher auf den gegebenen Beispielen brauchen.

(4) *Verfahren mit korrekten Ergebnissen*: Die Ergebnisse müssen in dem Sinne korrekt sein, dass alle Straßen abgedeckt werden, nur an Kreuzungen oder Enden von Sackgassen gewendet wird, höchstens 5 Tage benutzt werden, und alle Tagesstrecken an der Zentrale beginnen und enden. (Hier wird nicht verlangt, dass die Länge der Tour minimal ist.) Sofern Einsendungen kürzere Tagesstrecken als die angegebene untere Schranke vorschlagen, ist eines dieser Kriterien sicher verletzt.

(5) *Verfahren mit guter Ergebnisqualität*: Die Länge der längsten Tagestour soll möglichst kurz sein. Die Ergebnisse der vorgestellten Musterlösungen sind bereits sehr gut und ähnliche Ergebnisse können Pluspunkte geben. Falls die Heuristik zu einfach ist und dadurch die Ergebnisse deutlich schlechter als der zweite Greedy-Algorithmus (G2 in der Tabelle) sind, gibt es Punktabzug; starke Abzüge gibt es bei Ergebnissen, die schlechter sind als G1.
Dabei darf aber durchaus eine Abwägung zwischen Laufzeit und Ergebnisqualität getroffen werden: Wenn die Laufzeit des Verfahrens besonders gering ist, dürfen die Ergebnisse etwas schlechter sein.

(6) *Mehrere Lösungsansätze*: Da Heuristiken offensichtlich nicht zu garantiert optimalen Ergebnissen führen, ist es sinnvoll, sich über unterschiedliche Lösungsansätze zumindest Gedanken zu machen. Wer wirklich mehrere und deutlich unterschiedliche Ansätze ausführlich beschrieben oder sogar realisiert und miteinander verglichen hat, hat Pluspunkte verdient.

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Es muss erkannt werden, wenn das verwendete Verfahren keine optimalen Ergebnisse liefert. Soweit es nicht offensichtlich ist, sollte außerdem darauf eingegangen werden, warum das Verfahren korrekte Ergebnisse liefert, also alle Straßen tatsächlich abgedeckt werden.

(2) *Gute Überlegungen zur Laufzeit des Verfahrens*: Typischerweise sollte die asymptotische Laufzeit des Verfahrens betrachtet werden. Es ist ideal, aber nicht notwendig, hierfür die $\mathcal{O}$-Notation zu verwenden. Falls die Laufzeit im schlechtesten Fall exponentiell ist, sollte dies erkannt werden. Praktische Laufzeitmessungen können die Angabe der asymptotischen Laufzeit ersetzen. In Kombination – insbesondere, wenn die tatsächliche Laufzeit

auf den Beispielen stark von der Worst-Case-Laufzeit abweicht – kann ein Vergleich der asymptotischen und tatsächlichen Laufzeit auch Pluspunkte geben.

(3) *NP-Schwere des Problems erkannt*: Falls die NP-Schwere erkannt und korrekt begründet wird, etwa durch eine (informelle) Reduktion auf ein bekanntes NP-schweres Problem, können Pluspunkte vergeben werden.

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Es muss die berechnete maximale Länge der Tagesstrecken zu allen Beispielen vorhanden sein. Eine Lösung für `muellabfuhr0.txt` ist nicht nötig, da diese von BWINF bereits angegeben wurde.
Für alle Beispiele bis `muellabfuhr4.txt` sollten außerdem die konkreten Tagestouren dokumentiert werden. Für die übrigen Beispiele wird dies nicht gefordert.

(5) *Ergebnisse nachvollziehbar dargestellt*: Entweder die *Länge jeder Tagestour* oder aber die *Länge der längsten Tagestour* muss explizit dokumentiert sein. Es ist nicht ausreichend, nur die Touren selbst ohne Länge anzugeben.
Eine Tagestour selbst kann zum Beispiel als Liste von besuchten Knoten ausgegeben werden.

Aufgabe: Rechenrätsel

Bei dieser Aufgabe soll für eine gegebene Anzahl n von Operatoren (also $n+1$ Ziffern) eine Folge $a_0, a_1, a_2, \ldots, a_n$, $a_i \in \{0,1,2,\ldots,9\}$ und eine positive ganze Zahl b gefunden werden, sodass das Rätsel

$$a_0 \circ_1 a_1 \circ_2 \cdots \circ_n a_n = b$$

genau eine Lösung hat, wobei jedes $\circ_i$ durch ein Element der Menge $O := \{+,-,\cdot,:\}$ ersetzt werden soll.

2.1 Lösungsidee

Da b groß werden kann und es viele Möglichkeiten gibt b auszuwählen, bietet es sich an, a_0 bis a_n zu fixieren und für $b = 1,2,3,\ldots$ zu prüfen, ob das so entstandene Rätsel genau eine Lösung besitzt oder nicht. Für eine fixe Folge der a_i können wir dazu die Multimenge M der erreichbaren Zahlen b betrachten, in der jedes b so oft vorkommt, wie es mögliche Belegungen der $\circ_i$ mit Elementen aus O gibt. Haben wir M gefunden, können wir mit linearer Suche prüfen, welche Zahlen dort genau einmal vorkommen. Die Folge der a_i soll dabei vorher schon zufällig (mit bestimmten Einschränkungen, siehe unten) bestimmt werden.

Dass dies funktioniert, kann veranschaulicht werden, wenn man sich beim Beispielrätsel

$$4 \circ 3 \circ 2 \circ 6 \circ 3 \circ 9 \circ 7 \circ 8 \circ 2 \circ 9 \circ 4 \circ 4 \circ 6 \circ 4 \circ 4 \circ 5$$

die Liste der kleinsten eindeutig darstellbaren positiven ganzen Zahlen anschaut:

$$4792, 5118, 5136, 5138, 5142, 5152, 5156, 5163, 5167, 5168, 5194, 5200, 5213, 5215$$

Schaut man sich andere Rätsel an, ergeben sich ähnliche Listen. Dies liegt daran, dass die kleinen Zahlen sich fast immer auf mehrere Arten darstellen lassen. Aber ab 15 Operatoren gibt es zum Beispiel ca. 4000 bis 8000 viele Lösungen für b, die auch schnell größer werden. Wünschenswert wären kleine Ergebnisse, damit das Rätsel möglichst interessant wird und einfache Lösungswege über Blöcke von Multiplikationen, um überhaupt die richtige Größenordnung zu erreichen, ausgeschlossen werden. Somit werden wir die kleinste mögliche Lösung für b wählen.

„Uninteressante“ und „Interessante“ Rätsel

In der Aufgabenstellung ist die Bedingung genannt, dass die Rätsel „interessant und unterschiedlich“ sein sollen. Dies schließt etwa aus, bewusst $b = \prod_{i=0}^{n} a_i$ zu wählen. Für $n \geq 2$ und $a_i \geq 2$ lässt sich nämlich mit einem Größenargument beweisen, dass $\circ_1 = \circ_2 = \cdots = \circ_n = \cdot$ dann die einzige Lösung wäre.

Es gibt auch Rätsel, bei denen es aus einem *zahlentheoretischen* Grund, also etwa wegen Teilbarkeits- oder Restklassenbetrachtungen, genau eine Lösung gibt. Ein einfaches Beispiel hierfür ist $9 \circ_1 9 \circ_2 3 \circ_3 1 = 83$. Eine Größenbetrachtung schließt $\circ_1 = :$ aus, durch eine Betrachtung modulo 3 sieht man ein, dass $\circ_3$ durch $-$ ersetzt werden soll, mit modulo 9, dass $\circ_2$ durch $+$ ersetzt werden soll. Dies ergibt die einzig mögliche Lösung $9 \cdot 9 + 3 - 1 = 83$.

Durch eine Kombination dieser beiden Ansätze können sehr effizient sehr große Rätsel produziert werden, für die die Lösung allerdings ebenfalls schnell in linearer oder höchstens quadratischer Zeit durch sukzessive Größen- und Modulobetrachtungen gefunden werden kann.

Stattdessen wollen wir Rätsel erzeugen, für deren Lösung es eher Brute-Force-ähnlicher Ansätze bedarf. Dafür gehen wir davon aus, dass solche Rätsel am ehesten erstellt werden können, indem das Programm, das die Rätsel erstellt, ebenfalls nur Brute-Force-ähnliche Ansätze verwendet, auch wenn dabei durchaus Optimierungen möglich sind (die wir auch umsetzen werden).

Grundlegender Ansatz

Wir können einfach jede mögliche Belegung der $\circ_i$ testen und jeweils den Wert von b berechnen, den man dadurch erhält. Insgesamt wird so die Multimenge M aller möglichen Rechenergebnisse berechnet. Dabei müssen alle $\mathcal{O}(4^n)$ Belegungen ausprobiert und jedes Mal das Ergebnis berechnet werden, was $\mathcal{O}(n)$ Zeit in Anspruch nimmt. Werden die möglichen Ergebnisse in einer Multimenge verwaltet, kann diese bis zu 4^n Elemente haben. Das Einfügen einer Zahl in diese Multimenge benötigt Zeit $\mathcal{O}(\log(4^n)) = \mathcal{O}(n)$. Mit jeder Belegung muss also ein Ergebnis berechnet und einmal in die Multimenge eingefügt werden, woraus sich eine Laufzeit von $\mathcal{O}(4^n)\,(\mathcal{O}(n)+\mathcal{O}(n)) = \mathcal{O}(4^n \cdot n)$ ergibt.

Verbesserter Ansatz mit Dynamischer Programmierung

Das Hauptproblem beim grundlegenden Ansatz ist, dass oft der gleiche Wert für b gefunden wird. Es ist egal, ob man eine Zahl b mit 2 oder 2000 verschiedenen Belegungen der $\circ_i$ erhält. Kann man zum Beispiel mit den ersten k ($k < n$) Operatoren $\circ_1$ bis $\circ_k$ ein Zwischenergebnis schon auf zwei verschiedene Arten darstellen, muss mit einem dritten gleichen Zwischenergebnis nicht weiter gerechnet werden. Es ist nur die Information wichtig, dass man mit diesem Zwischenergebnis keine eindeutige Lösung erhält. Anders ausgedrückt: es muss nicht mit jeder Belegung von $\circ_1$ bis $\circ_k$, die zum gleichen Zwischenergebnis b_k führt, separat weiter gerechnet werden, da die gleichen Rechnungen ausgeführt werden.

Weiterhin kommt es vor, dass die gleichen Zwischenergebnisse immer wieder berechnet werden. Zum Beispiel das Produkt zweier benachbarter Zahlen.

Um diese beiden Probleme zu beheben, wird ein verbesserter Ansatz mit dynamischer Programmierung eingeführt.

Die Berechnung von Zwischenergebnissen soll in der gleichen Reihenfolge wie die tatsächliche Berechnung von b erfolgen, also Punkt vor Strich. Sei also zunächst $P_{i,j}$ die Multimenge der positiven ganzen Zahlen b, die

$$a_{i-1} \circ_i a_i \circ_{i+1} \cdots \circ_j a_j$$

als Wert annehmen kann, wenn $\circ_i$ bis $\circ_j$ durch $\cdot$ oder : ersetzt werden. Offenbar gilt $P_{i+1,i} = \{a_i\}$ für $i = 0, 1, \ldots, n$. Ferner kann $P_{i,j}$ aus $P_{i,j-1}$ mithilfe der Rekursion

$$P_{i,j} = \left\{ x a_j \,|\, x \in P_{i,j-1} \right\} \uplus \left(\left\{ \frac{x}{a_j} \,|\, x \in P_{i,j-1} \right\} \cap \mathbb{Z} \right)$$

für $1 \leq i \leq j \leq n$ berechnet werden.[8] Ganz analog lässt sich S_i, die Multimenge der möglichen Summen/Differenzen von Produkten berechnen. Formal sei S_i die Multimenge der ganzen Zahlen b, die

$$a_0 \circ_1 a_1 \circ_2 \cdots \circ_i a_i$$

als Wert annehmen kann, wenn $\circ_1$ bis $\circ_i$ jeweils durch ein beliebiges Element von O ersetzt werden. Die Berechnung von S_j aus S_0 bis S_{j-1} kann also als Vereinigung der Verknüpfungen von jeweils S_{i-1} mit $P_{i,j}$ erfolgen, wobei i die Werte 1 bis j annimmt. In Formeln ausgedrückt wird also $S_0 = \{a_0\}$ und dann

$$S_j = P_{0,j} \uplus \left(\biguplus_{i=1}^{j} \left(\{s+p \,|\, s \in S_{i-1},\, p \in P_{i,j}\} \uplus \{s-p \,|\, s \in S_{i-1},\, p \in P_{i,j}\} \right) \right)$$

für $j = 1, 2, 3, \ldots$ berechnet. So erhält man zuletzt die gesuchte Multimenge S_n, in der nach dem kleinsten positiven Element gesucht werden kann, das dort genau einmal vorkommt. Am kleinen Beispiel aus der Aufgabenstellung

$$4 \circ 4 \circ 3 = 13$$

kann dies so verdeutlicht werden:

j	0	1	2
$P_{0,j}$	$\{4\}$	$\{1, 16\}$	$\{3, 48\}$
$P_{1,j}$		$\{4\}$	$\{12\}$
$P_{2,j}$			$\{3\}$
S_j	$\{4\}$	$\{0, 1, 8, 16\}$	$\{-8, -3, -2, 3, 3, 4, 5, 11, 13, 16, 19, 48\}$

Man sieht, dass 13 zum Beispiel eindeutig darstellbar ist.

Das Problem, dass Zwischenergebnisse mehrfach berechnet werden, kann mit diesem Ansatz auch überwunden werden. Identische Elemente in S_i bzw. $P_{i,j}$ erzeugen immer identische Ergebnisse für S_n. Also kann jede Zahl, die mehr als zweimal in $P_{i,j}$ bzw. S_i vorkommt, so weit gestrichen werden, bis sie dort nur zweimal vorhanden ist, bzw. nur eingefügt werden, wenn dieses Element bisher nur höchstens einmal enthalten ist.

Mit diesem Ansatz wird auch im schlimmsten Fall keine schlechtere Laufzeit als mit dem grundlegenden Ansatz erzielt. Die Multimenge S_k hat höchstens 4^k Elemente, S_0 bis S_n haben also insgesamt höchstens

$$\sum_{i=0}^{n} 4^i = \frac{4^{n+1} - 1}{3} \in \mathcal{O}(4^n)$$

Elemente, mit $\mathcal{O}(n)$ Zeit für das Einfügen haben wir hier eine Laufzeit von höchstens $\mathcal{O}(4^n \cdot n)$ und somit eine Gesamtlaufzeit von $\mathcal{O}(2^n \cdot n^3 + 4^n \cdot n) = \mathcal{O}(4^n \cdot n)$. Anders gesagt: Es wird nie mehr als beim grundlegenden Ansatz berechnet, aber es kann bedeutsame Einsparungen geben, da im Vergleich zu den (Zwischen-)Ergebnissen sehr kleine Ziffern und viele Dopplungen gestrichen und Berechnungen gespart werden können.

[8] Wir benutzen $\uplus$ als Symbol für die Vereinigung von Multimengen.

Wahl der a_i

Bisher haben wir nichts über die Wahl der Werte der a_i gesagt und diese beliebig gewählt, laut Aufgabenstellung aus der Menge $\{0,1,2,\ldots,9\}$. Um das Rätsel jedoch *interessant* zu halten, wird auf die Ziffern 0 und 1 verzichtet. Durch 0 darf man überhaupt nicht teilen und bei diesen Ziffern entstehen schnell mehrere Lösungen für viele kleine Ergebnisse. So ist es etwa egal, ob 0 addiert oder subtrahiert, 1 multipliziert oder dividiert und ob ein Produkt oder ein Quotient mit 0 multipliziert wird.

Es ist auch möglich, ein wenig Rechenzeit zu sparen, indem bevorzugt kleine a_i oder aufeinanderfolgende Zahlen möglichst teilerfremd gewählt werden. Hier werden die a_i unabhängig voneinander mit folgenden empirisch bestimmten Wahrscheinlichkeiten gewählt:

Wert	2	3	4	5	6	7	8	9
Wahrscheinlichkeit	27,0%	21,6%	16,2%	10,8%	8,1%	6,5%	5,4%	4,3%

Im Vergleich zur gleichverteilten Wahl läuft so das Programm bei ca. 20 Operatoren ungefähr doppelt so schnell und liefert wesentlich kleinere Rätselergebnisse. Nachteil wiederum ist, dass die mit höherer Wahrscheinlichkeit gewählten Ziffern entsprechend häufig in den Rätseln auftauchen, wodurch diese als weniger „interessant“ angesehen werden können.

2.2 Beispiele

Zum Beispiel werden diese Rätsel gefunden:

- 3 Operatoren: $2 \circ 3 \circ 2 \circ 6 = 6$, gefunden in 0,00 s
- 5 Operatoren: $4 \circ 5 \circ 4 \circ 7 \circ 2 \circ 5 = 11$, gefunden in 0,00 s
- 8 Operatoren: $4 \circ 3 \circ 2 \circ 4 \circ 3 \circ 3 \circ 2 \circ 3 \circ 3 = 110$, gefunden in 0,00 s
- 10 Operatoren: $4 \circ 9 \circ 3 \circ 7 \circ 7 \circ 3 \circ 2 \circ 3 \circ 2 \circ 3 \circ 8 = 353$, gefunden in 0,00 s
- 12 Operatoren: $2 \circ 3 \circ 2 \circ 9 \circ 8 \circ 5 \circ 9 \circ 7 \circ 2 \circ 7 \circ 8 \circ 3 \circ 8 = 1521$, gefunden in 0,02 s
- 15 Operatoren: $6 \circ 5 \circ 4 \circ 7 \circ 6 \circ 9 \circ 3 \circ 3 \circ 3 \circ 3 \circ 7 \circ 8 \circ 2 \circ 3 \circ 6 \circ 3 = 6295$, gefunden in 0,19 s
- 18 Operatoren: $4 \circ 3 \circ 4 \circ 2 \circ 8 \circ 3 \circ 6 \circ 3 \circ 4 \circ 3 \circ 8 \circ 3 \circ 7 \circ 7 \circ 2 \circ 3 \circ 2 \circ 7 \circ 4 = 17464$, gefunden in 1,17 s
- 20 Operatoren: $2 \circ 7 \circ 6 \circ 9 \circ 2 \circ 3 \circ 2 \circ 2 \circ 2 \circ 3 \circ 2 \circ 3 \circ 8 \circ 3 \circ 6 \circ 3 \circ 2 \circ 7 \circ 4 \circ 4 \circ 2 = 19330$, gefunden in 1,67 s

2.3 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Es sind alle internen Darstellungen von Rätseln akzeptabel, die das Generieren nicht unnötig verkomplizieren. Die Verwendung von Strings ist eher keine gute Idee.

(2) *Laufzeit des Verfahrens in Ordnung*: Die asymptotische Laufzeit des Programms, sofern analytisch charakterisierbar, sollte die des grundlegenden Ansatzes keinesfalls überschreiten. Verbesserungen (oder auch alternative Verfahren) sollten ermöglichen, dass auch für 15 oder sogar mehr Operanden in wenigen Sekunden eine Lösung gefunden wird. Verfahren mit besonders kurzen Laufzeiten können mit Pluspunkten belohnt werden, sofern sie nicht erheblich leichter zu lösende Rätsel generieren als aufwändigere Verfahren. Verfahren mit polynomiellen Laufzeiten, die zu leichten Rätseln führen, sind dieses Mal hingegen nicht akzeptabel. Wenn das gewählte Verfahren im Prinzip dem grundlegenden Ansatz entspricht, werden 4 Punkte abgezogen.

(3) *Verfahren nicht unnötig ineffizient*: Wird zuerst ein Rätsel inklusive Operatoren erzeugt und dann durch Probieren aller anderen Belegungen die Eindeutigkeit überprüft, werden 2 Punkte abgezogen. Es werden Verbesserungen erwartet, insbesondere das Vermeiden mehrfacher Berechnungen.

(4) *Verfahren mit korrekten Ergebnissen*: Die Bedingungen der Aufgabenstellung (insbesondere Punkt vor Strich ohne Klammern, Grundrechenarten, ganzzahlige Zwischenergebnisse, eindeutiges nicht-negativ ganzzahliges Endergebnis) sollten richtig verstanden und umgesetzt werden. Eindeutig heißt hier, dass es nur eine mögliche Belegung der $\circ$ gibt, die diese Bedingungen erfüllt und zum richtigen Ergebnis führt.
Wenn das Ergebnis nicht eindeutig ist, so werden bis zu 4 Punkte abgezogen. Wenn die Bedingungen so interpretiert wurden, dass auch negative Zwischenergebnisse ausgeschlossen werden, soll es dafür keinen Punktabzug geben.
Sollte eine Interpretation der Bedingungen zu leichten Einschränkungen bei den Ergebnissen führen, so ist dies akzeptabel, wenn es dokumentiert ist.

(5) *Wahl der Operanden und der Lösung geeignet*: Es sollte erkannt werden, dass die Zahlen und Ergebnisse frei gewählt werden können, und dies für die benötigte Variabilität auch getan werden. Es bietet sich an, die a_i frei zu wählen. Wenn bestimmte Zahlen, wie 0 oder 1, von vornherein ausgeschlossen werden, so muss dies begründet werden. Wenn das Ergebnis b so gewählt wird, dass das Rätsel damit schwieriger wird (zum Beispiel wenn kleine b bevorzugt werden) und diese Beobachtung erwähnt wird, gibt es dafür 2 Pluspunkte.

(6) *Rätsel interessant und unterschiedlich*:
- Laut Aufgabenstellung sollten die erzeugten Rätsel „interessant und unterschiedlich“ sein. Es ist also zum Beispiel nicht in Ordnung, das Ergebnis des Rätsels einfach als das Produkt der Operanden zu wählen. Kriterien und Bedingungen für die Erstellung „interessanter“ Rätsel sollten genannt und begründet werden. Eine zufällige Wahl der Operanden und Operatoren reicht aus, um das Kriterium „interessant“ zu befriedigen, dies sollte allerdings begründet werden. Wenn das Kriterium nicht explizit erwähnt bzw. begründet wird, so wird ein Punkt abgezogen.
- Rein deterministische Ansätze, die nur von der geforderten Anzahl an Operatoren ab-

hängen, sind ungeeignet, da *unterschiedliche* Rätsel benötigt werden.

(7) *Mehrere Lösungsansätze*: Wer mehrere und deutlich unterschiedliche Ansätze ausführlich beschrieben oder sogar realisiert und miteinander verglichen hat, hat Pluspunkte verdient.

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Die Korrektheit des Verfahrens bezüglich der an die Rätsel gestellten Anforderungen sollte möglichst gut begründet werden, insbesondere die eindeutige Lösbarkeit der generierten Rätsel. Falls ein Algorithmus mit exponentieller Laufzeit gewählt wurde, sollten ebenfalls Gründe genannt werden; formale Argumente können Pluspunkte bringen. Bei der Wahl eines schnelleren Ansatzes sollte auf mögliche Nachteile bei der Generierung der Rätsel eingegangen werden.

(2) *Gute Überlegungen zur Laufzeit des Verfahrens*: Die Laufzeit kann bei einigen Optimierungen nur schwer konkret abgeschätzt werden, allerdings wird erwartet, dass man mit einfachen Überlegungen zeigt, warum der Ansatz besser ist als etwa der grundlegende. Auch hier können für formale Argumente Pluspunkte vergeben werden.

(3) *Weitere gute analytische Überlegungen*: Es wird ein Verfahren verlangt, das den grundlegenden Ansatz verbessert. Hierfür sind vielerlei Optimierungen möglich, nicht nur bei der Berechnung, sondern auch bei der Wahl der Operanden. Hier können vor allem empirische Beobachtungen dazu ausgenutzt werden, wann möglichst schnell ein eindeutiges Ergebnis gefunden wird.

3. Dokumentation

(3) *Erzeugte Rätsel dokumentiert*: Zwar gibt es bei dieser Aufgabe keine vorgegebenen Beispiele, aber es sollten natürlich, je nach Ansatz, beispielhaft mindestens sechs generierte Rätsel mit jeweils verschiedener Operatorenanzahl, darunter die Anzahlen 2 und 3, gezeigt werden. Besser sind solche Rätsel, die repräsentativ für Ausgaben des Programms sind. Für eine erläuterte Dokumentation weiterer, die Leistungsfähigkeit des Rätselgenerators aussagekräftig demonstrierender Beispiele kann es Pluspunkte geben.

(4) *Ergebnisse nachvollziehbar dargestellt*: Für alle dokumentierten Beispielrätsel müssen die Lösungen mit angegeben werden.

Aufgabe: Hex-Max

3.1 Lösungsidee

Wir möchten die größte Hex-Zahl finden, die mit maximal m Umlegungen aus einer gegebenen Hex-Zahl erzeugt werden kann.

Im Folgenden bezeichnen wir die ursprüngliche, gegebene Hex-Zahl mit A und mit A_i ($1 \leq i \leq n$) die i-te der n Ziffern. Wir beschreiben A_i als Menge, die die Nummern aller mit Stäbchen belegten Positionen (auch Segmente genannt; siehe Abbildung 3.1 für die von uns gewählte Nummerierung der Positionen) enthält: $A_i = \{1,2,5,6\}$ entspräche z. B. der Ziffer 4, $A_i = \{2,5,6\}$ wäre keine Ziffer des Hexadezimalsystems. Aus A erzeugte Hex-Zahlen bezeichnen wir mit anderen Großbuchstaben wie B und einzelne Ziffern manchmal mit Kleinbuchstaben wie b; die Notation gilt entsprechend.

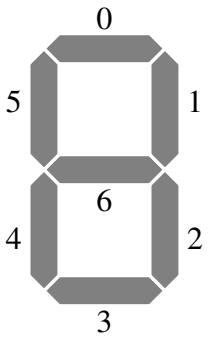

Abbildung 3.1: Nummerierung der Positionen

Bei $A_i \setminus B_i$ handelt es sich daher um die Menge der Positionen, die in Ziffer A_i mit einem Stäbchen belegt sind, aber in B_i nicht. $|A_i \setminus B_i|$ und $|B_i \setminus A_i|$ sind darum die Anzahl der Stäbchen, die entfernt bzw. abgelegt werden müssen, um B_i aus A_i zu erhalten.

Wir verwenden die folgenden beiden Lemmata (mathematische Aussagen kleinen Umfangs – Hilfssätze – mit Beweis), um das Problem etwas umzuformulieren.

Lemma 1. *Wenn eine Hex-Zahl B aus der ursprünglichen Hex-Zahl A durch das Anheben von l Stäbchen und das Ablegen von l Stäbchen an bisher freien Positionen erzeugt werden kann, so kann sie auch durch das Umlegen von höchstens l Stäbchen im Sinne der Aufgabenstellung erzeugt werden, d. h. durch das einzelne Bewegen von höchstens l Stäbchen, ohne dass die Darstellung einer Ziffer dabei komplett geleert würde.*

Beweis. Wir betrachten den folgenden Algorithmus:

Bei der i-ten Ziffer müssen $|A_i \setminus B_i|$ Stäbchen angehoben und $|B_i \setminus A_i|$ Stäbchen abgelegt werden. Zunächst bewegen wir daher innerhalb jeder i-ten Ziffer $\gamma_i := \min(|A_i \setminus B_i|, |B_i \setminus A_i|)$ Stäbchen von ihrer bisherigen Position an eine zu belegende. Da wir nur innerhalb der Ziffer Stäbchen umlegen, kann die Ziffer dabei nicht geleert werden. Nun liegen noch $\alpha_i := |A_i \setminus B_i| - \gamma_i$ Stäbchen bei Ziffer i, die von dort entfernt werden müssen, und zu $\beta_i := |B_i \setminus A_i| - \gamma_i$ Positionen bei Ziffer i müssen noch Stäbchen bewegt werden. Man beachte, dass $\alpha_i = 0$ oder $\beta_i = 0$ gilt; an jeder Ziffer müssen also nur noch Stäbchen entfernt oder nur noch Stäbchen abgelegt werden. Damit kann auch die Bedingung, dass keine Darstellung einer Ziffer komplett geleert werden darf, nicht verletzt werden: Durch das Ablegen eines Stäbchens ohnehin nicht, und wenn wir von einer Ziffer lediglich Stäbchen entfernen und am Ende wieder eine (nicht leere) Ziffer erhalten, kann die Darstellung nicht dazwischen geleert gewesen sein.

Wir nehmen nun von einer beliebigen Ziffer, von der noch Stäbchen entfernt werden müssen ($\alpha_i > 0$), ein solches Stäbchen und legen es bei einer Ziffer, bei der noch Positionen belegt werden müssen ($\beta_i > 0$), an einer solchen ab. Das wiederholen wir, bis keine Stäbchen mehr angehoben bzw. abgelegt werden müssen.

Wir haben offensichtlich nicht mehr Stäbchen bewegt als mindestens angehoben bzw. abgelegt werden müssen, um B aus A zu erhalten, und damit insbesondere höchstens l Stäbchen. □

Lemma 2. *Lässt sich eine Hex-Zahl B aus der ursprünglichen Hex-Zahl A mit höchstens m Umlegungen erzeugen, so kann sie auch durch das Anheben von l Stäbchen und das Ablegen von l Stäbchen erzeugt werden, mit $l \leq m$.*

Beweis. Wenn B aus A mit $o \leq m$ Umlegungen erzeugt wird, sind danach höchstens o Positionen mit einem Stäbchen belegt, die davor nicht belegt waren ($\sum_i |B_i \setminus A_i| \leq o$), und höchstens o Positionen sind danach nicht mehr belegt, waren es aber davor ($\sum_i |A_i \setminus B_i| \leq o$). Da die Gesamtanzahl der Stäbchen gleich bleibt, sind beide Anzahlen gleich: $l := \sum_i |B_i \setminus A_i| = \sum_i |A_i \setminus B_i| \leq o \leq m$. Wenn wir also die l Stäbchen, die in A an Positionen liegen, die in B nicht mehr belegt sind, anheben, und dann an den Positionen ablegen, die in B belegt sind, aber in A noch frei, erhalten wir B auf die gewünschte Weise. □

Wir wollen daher nun folgendes Problem lösen: Was ist die größte Hex-Zahl, die durch das Anheben von l Stäbchen und das Ablegen von l Stäbchen erzeugt werden kann, wobei $l \leq m$ sein muss? Wenn wir diese Hex-Zahl bestimmen können, können wir mit dem im Beweis von Lemma 1 skizzierten Algorithmus auch eine gültige Folge von Umlegungen finden, die sie erzeugt; außerdem bezieht dies nach Lemma 2 jede Hex-Zahl mit ein, die auch durch gültige Umlegungen zu erhalten ist, wir bestimmen so also tatsächlich die gesuchte maximale Hex-Zahl.

Eine wichtige Beobachtung benötigen wir für alle unsere Algorithmen: Eine größere Ziffer an einer Stelle der Hex-Zahl macht die Zahl größer als jegliche Änderungen darauf folgender („less significant“) Stellen. Formal: Es gilt stets $B > B'$, wenn $B_1 = B'_1, \ldots, B_{i-1} = B'_{i-1}$ und $B_i > B'_i$, unabhängig von den Werten von $B_{i+1}, \ldots, B_n$ und $B'_{i+1}, \ldots, B'_n$. Daher liegt ein Greedy-Ansatz nahe, bei dem man die Ziffern von links nach rechts jeweils so groß wie möglich zu machen versucht. Dabei muss jedoch darauf geachtet werden, dass mit den dafür nötigen Veränderungen auch noch rechts von $i+1$ eine gültige Hex-Zahl erzeugt werden kann.

Dynamische Programmierung: Erster Ansatz

Wir definieren $r(a,b)$ als die Anzahl der Stäbchen, die übrig bleiben, wenn man die Ziffer a zur Ziffer b ändert; $r(a,b)$ kann negativ sein, dann benötigt man zusätzliche Stäbchen, um a zu b zu ändern. Außerdem sei $s(a,b)$ die Anzahl der Positionen von Stäbchen, die sich ändern, wenn man a zu b ändert. Es gilt also

$$r(a,b) := |a \setminus b| - |b \setminus a| \text{ und}$$
$$s(a,b) := |a \setminus b| + |b \setminus a|.$$

Im Folgenden bezeichnen wir mit $[A_1 \cdots A_i]$ eine Hex-Zahl mit $1 \leq i \leq n$ Ziffern $A_1, \ldots, A_i$.

Wir möchten die folgende Funktion berechnen:

$$dp : \{0,\ldots,n\} \times \{0,\ldots,m\} \times \{-m,\ldots,m\} \to \{\text{Hex-Zahlen}\} \cup \{-\infty\},$$
$$(i,x,y) \mapsto \begin{cases} \text{Größte Hex-Zahl } [B_1 \cdots B_i] \text{ mit } i \text{ Stellen,} \\ \text{die an genau } x \text{ Positionen anders als } [A_1 \cdots A_i] \text{ ist und} \\ y \text{ mehr Stäbchen als } [A_1 \cdots A_i] \text{ enthält.} \end{cases}$$

Dabei darf y auch negativ sein; in dem Fall handelt es sich um eine Hex-Zahl mit weniger Stäbchen als $[A_1 \cdots A_i]$. Für jede Kombination von i,x,y, für die es keine solche Hex-Zahl gibt, setzen wir $dp(i,x,y) := -\infty$.

Wir stellen fest, dass $\max_{0\leq x\leq 2m} dp(n,x,0)$, also die größte Hex-Zahl mit höchstens $2m$ geänderten Positionen und unveränderter Anzahl Stäbchen, gerade die gesuchte größte Hex-Zahl nach Anheben von $l \leq m$ Stäbchen und Ablegen von l Stäbchen ist. Man beachte, dass zwangsläufig $dp(n,x,0) = -\infty$ für ungerade x gilt: Wenn die Anzahl der Stäbchen gleich geblieben ist, muss von ebenso vielen Positionen ein Stäbchen entfernt worden sein, wie Positionen mit einem belegt wurden, d. h. die Summe dieser beiden gleichen Anzahlen ist gerade.

Außerdem ist $dp(0,0,0) = []$ die aus 0 Ziffern bestehende „leere“ Hex-Zahl.

Wie berechnen wir also diese Funktion? Es gilt die Rekursionsgleichung

$$dp(i,x,y) = \max_{b\in\mathscr{H}} \left[\underbrace{dp(i-1, x-s(A_i,b), y-r(A_i,b))}_{\text{Bisherige Hex-Zahl mit } i-1 \text{ Ziffern}} \underbrace{b}_{\text{Neue Ziffer}} \right]$$

für alle i,x,y, wobei $\mathscr{H} := \{\texttt{0}, \texttt{1}, \ldots, \texttt{e}, \texttt{f}\}$ die Menge der 16 Hex-Ziffern ist.

Wir können also für $i = 1,2,\ldots,n$ jeweils die Werte von $dp(i,x,y)$ für alle x,y berechnen und müssen dabei jeweils nur auf (schon berechnete) Werte von $dp(i-1,\cdot,\cdot)$ zurückgreifen.

Beim Berechnen eines Wertes der Funktion müssen 16 Möglichkeiten für b probiert werden; der Vergleich der Hex-Zahlen („Ist die aktuelle Hex-Zahl größer oder kleiner als die bisher größte gefundene?“) benötigt jeweils $i \leq n$ Operationen. Es gibt $(n+1)(m+1)(2m+1) \in \mathscr{O}(nm^2)$ Kombinationen von i,x,y, also ergibt sich eine Laufzeit von $\mathscr{O}(nm^2 \cdot 16 \cdot n) = \mathscr{O}(n^2m^2)$.[9]

Dynamische Programmierung: Zweiter Ansatz

Wir stellen einen zweiten Algorithmus, mit deutlich geringerer Laufzeit, vor.

Wir möchten dazu zunächst die folgende Funktion vorberechnen:

$$e : \{0,\ldots,n\} \times \{-m,\ldots,m\} \to \{0,\ldots,7n,\infty\},$$
$$(i,y) \mapsto \min\left\{ \sum_{i<j\leq n} s(A_j,b_j) \,\middle|\, b_{i+1},\ldots,b_n \in \mathscr{H}, y = \sum_{i<j\leq n} r(A_j,b_j) \right\},$$

die minimal nötige Anzahl an veränderten Stäbchen-Positionen in den Ziffern $A_{i+1},\ldots,A_n$, um von dort genau y Stäbchen übrig zu haben, wobei diese Ziffern wieder eine gültige Hex-Zahl ergeben sollen. Ist dies nicht möglich, definieren wir $e(i,y) := \infty$.

Dazu verwenden wir

$$\begin{aligned} e(n,0) &= 0, \\ e(n,y) &= \infty \text{ für } y > 0 \text{ und die Rekursionsgleichung} \\ e(i,y) &= \min_{b\in\mathscr{H}} e(i+1, y-r(A_{i+1},b)) + s(A_{i+1},b) \text{ für } i < n, y \end{aligned}$$

(mit $e(\cdot,y) := \infty$ für alle $y \notin \{-m,\ldots,m\}$).

Wir können die größte erreichbare Hex-Zahl bestimmen, indem wir von links nach rechts jeweils die größte Ziffer finden und wählen, mit der sich auch rechts der aktuellen Position noch eine gültige Hex-Zahl bilden lässt.

[9] Falls $m > \frac{7}{2}n$, können wir m durch $\frac{7}{2}n$ ersetzen (denn wir werden ein Stäbchen nie mehrfach umlegen), womit die Laufzeit sogar in $\mathscr{O}(n^2 \min(n,m)^2)$ liegt.

Wenn vor Stelle i bisher x Positionen von Stäbchen (bei Stelle $j < i$) geändert wurden und y Stäbchen „fehlten“, also von Stelle $j \geq i$ geholt werden müssen, ist

$$\max\{b \in \mathscr{H} | x + s(A_i, b) + e(i, y - r(A_i, b)) \leq 2m\}$$

die größte Ziffer, die an Stelle i gelegt werden kann. Dies lässt sich in folgenden *Greedy-Algorithmus* umsetzen:

Algorithmus 1 Greedy-Algorithmus zum Berechnen der größten erreichbaren Hex-Zahl

```
Berechne e vor
x ← 0                                   ▷ Anzahl geänderter Positionen
y ← 0                                   ▷ Anzahl fehlender Stäbchen
for i = 1,2,...,n do
    for b = f,e,...,1,0 do              ▷ Wähle greedy die größtmögliche Ziffer
        x' ← x + s(A_i, b)              ▷ Neue Anzahl geänderter Positionen
        y' ← y − r(A_i, b)              ▷ Neue Anzahl fehlender Stäbchen
        if x' + e(i, y') ≤ 2m then      ▷ Anzahl mind. nötiger Änderungen ≤ 2m?
            B_i ← b
            x ← x', y ← y'
            break
        end if
    end for
end for
return B
```

Algorithmus 1 hat abseits der Vorberechnung eine Laufzeit von nur $\mathscr{O}(n \cdot 16) = \mathscr{O}(n)$. In der Vorberechnung wird jedoch für alle $(n+1)(2m+1) \in \mathscr{O}(nm)$ Kombinationen von i, y jeweils in Laufzeit $\mathscr{O}(16) = \mathscr{O}(1)$ der Wert von $e(i, y)$ berechnet. Die Gesamtlaufzeit ist also in $\mathscr{O}(nm)$.[10]

Berechnen der einzelnen Umlegungen

Sobald wir die größte Hex-Zahl berechnet haben, die mit maximal m Umlegungen erzeugt werden kann, können wir den in Lemma 1 skizzierten Algorithmus anwenden, um eine gültige Folge von Umlegungen zu finden, die sie erzeugt. Wir zeigen im Folgenden eine effiziente Umsetzung dieses Algorithmus mit Laufzeit $\mathscr{O}(n)$. Die Queues q_1, q_2 dienen dazu, nacheinander Paare von belegten, zu leerenden und leeren, zu belegenden Positionen abzuarbeiten.

[10] Analog zu Fußnote 9 liegt die Laufzeit sogar in $\mathscr{O}(n \min(n, m))$.

Algorithmus 2 Berechnen einzelner Umlegungen, um B aus A zu erhalten

for all $1 \leq i \leq n$ **do**
  Lege γ_i Stäbchen von $A_i \setminus B_i$ nach $B_i \setminus A_i$ um
end for
$q_1 \leftarrow$ Queue, $q_2 \leftarrow$ Queue
for all $1 \leq i \leq n$ **do**
  for all $p \in A_i \setminus B_i$ **do**
    q_1.PUSH((i, p))
  end for
  for all $p \in B_i \setminus A_i$ **do**
    q_2.PUSH((i, p))
  end for
end for
while $|q_1| (= |q_2|) > 0$ **do**
  $(i_1, p_1) \leftarrow q_1$.POP()
  $(i_2, p_2) \leftarrow q_2$.POP()
  Lege Stäbchen von Position p_1 bei Ziffer i_1 zu Position p_2 bei Ziffer i_2 um
end while

3.2 Beispiele

Es folgen die Lösungen zu den gegebenen BWINF-Beispielen. Wie schon in der Beschreibung der Lösungsidee behandeln wir auch hier die Umlegungen gesondert und zeigen sie in einem eigenen, unteren Abschnitt.

Größte Hex-Zahl

Zunächst geben wir für jedes Beispiel die größte erreichbare Hex-Zahl an. Ziffern, die sich an mindestens einer Position geändert haben, sind rot markiert. Die Hex-Zahlen beginnen meist mit vielen `fs` (der größten Ziffer) und enden manchmal mit einigen `8`en (der Ziffer mit den meisten Stäbchen – da das `f` unterdurchschnittlich viele Stäbchen hat, liegt die Erwartung nahe, dass am Ende ein Überschuss an Stäbchen besteht); der Teil dazwischen ist unterstrichen.

hexmax0.txt

EE4

hexmax1.txt

FFFEA97B55

hexmax2.txt

FFFFFFFFF FFFFFFD9A9 BEAEE8EDA8 BDA989D9F8

hexmax3.txt

```
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFAA98BB 8B9DFAFEAE 888DD888AD 8BA8EA8888
```

hexmax4.txt

```
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFEB 8DE88BAA8A
DD888898E9 BA88AD9898 8F898AB7AF 7BDA8A61BA 7D4AD8F888
```

hexmax5.txt

```
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
FFFFFFFFFF FFFFFFFFFF F88EFA9EBE 89EFA99FBD AA8E8EAD88
AB899F8E8F 9AA9E9AD88 988EDA9A99 888EDAD989 A8BAFD8A88
8888888888 8888888888 8888888888 8888888888 8888888888
8888888888 8888888888 8888888888 8888888888 8888888888
8888888888 8888888888 8888888888 8888888888 8888888888
8888888888 8888888888 8888888888 8888888888 8888888888
8888888888 8888888888 8888888888 8888888888 8888888888
```

Einzelne Umlegungen

Wie vorgeschlagen geben wir nur für die Beispiele 0 bis 2 die einzelnen Umlegungen an; für die Beispiele 3 bis 5 belassen wir es bei der Ausgabe der jeweils größten erreichbaren Hex-Zahl. Im Folgenden zeigen wir jeweils den Zwischenstand nach jeder Umlegung.

hexmax0.txt

hexmax1.txt

```
509C43 1655
609C43 1655
6A9C43 1655
6AAC43 1655
FAAE43 1655
FFAE93 1655
FFFEA3 1655
FFFEA97655
```

hexmax2.txt

```
6326296 38F 1 18490 15A36CAEE2CdA06d4969 19F8
6926296 38F 1 18490 15A36CAEE2CdA06d4969 19F8
6A26296 38F 1 18490 15A36CAEE2CdA06d4969 19F8
6AP6296 38F 1 18490 15A36CAEE2CdA06d4969 19F8
6APA296 38F 1 18490 15A36CAEE2CdA06d4969 19F8
6APAP96 38F 1 18490 15A36CAEE2CdA06d4969 19F8
6APAPA6 38F 1 18490 15A36CAEE2CdA06d4969 19F8
6APAPAA 38F 1 18490 15A36CAEE2CdA06d4969 19F8
6APAPAA 98F 1 18490 15A36CAEE2CdA06d4969 19F8
6APAPAAA8F 1 18490 15A36CAEE2CdA06d4969 19F8
6APAPAAA8F7 18490 15A36CAEE2CdA06d4969 19F8
6APAPAAA8F= 18490 15A36CAEE2CdA06d4969 19F8
6APAPAAA8F=7 8490 15A36CAEE2CdA06d4969 19F8
6APAPAAA8F==8490 15A36CAEE2CdA06d4969 19F8
6APAPAAA8F==8590 15A36CAEE2CdA06d4969 19F8
6APAPAAA8F==8F90 15A36CAEE2CdA06d4969 19F8
6APAPAAA8F==8FA0 15A36CAEE2CdA06d4969 19F8
6APAPAAA8F==8FAA 15A36CAEE2CdA06d4969 19F8
AAPAPAAA8F==8FAA 15A36CAEE2CdA06d4969 19F8
FAPAPAAA8FF=8FAA 15A36CAEE2CdA06d4969 19F8
FFPAPAAA8FF=8FAA 15A36CAEE2CdA06d4969 19F8
FFFAPAAA8FFF8FAA 15A36CAEE2CdA06d4969 19F8
FFFFPAAA8FFF8FAA 15A36CAEE2CdA06d4969 19F8
FFFFFAAA8FFF8FAAd5A36CAEE2CdA06d4969 19F8
FFFFFAAA8FFF8FAAd5A36CAEE2CdA06d4969 19F8
FFFFFFAA8FFF8FAAd9A36CAEE2CdA06d4969 19F8
FFFFFFFA8FFF8FAAd9A96CAEE2CdA06d4969 19F8
FFFFFFFF8FFF8FAAd9A96EAEE2CdA06d4969 19F8
FFFFFFFFAFFF8FAAd9A96EAEE2CdA06d4969 19F8
FFFFFFFFAFFF8FAAd9A96EAEE8CdA06d4969 19F8
FFFFFFFFFFFF8FAAd9A96EAEE8EdA06d4969 19F8
FFFFFFFFFFFFAFAAd9A96EAEE8EdA86d4969 19F8
FFFFFFFFFFFFAAAd9A96EAEE8EdA86d9969 19F8
FFFFFFFFFFFFAAAd9A96EAEE8EdA86dA969 19F8
```

```
FFFFFFFFFFFFFFFAd9A9bEAEE8EdA8bdA989 !9F8
FFFFFFFFFFFFFFFAd9A9bEAEE8EdA8bdA989-19F8
FFFFFFFFFFFFFFFAd9A9bEAEE8EdA8bdA989J9F8
FFFFFFFFFFFFFFFFd9A9bEAEE8EdA8bdA989d9F8
```

3.3 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Die Ziffern der Siebensegmentanzeige müssen geeignet modelliert und im Programm dargestellt werden. Für die Lösungsbeschreibung eignet sich die Darstellung als Menge von belegten Positionen oder Bitset. Bei der Implementierung des ersten Aufgabenteils ist die Darstellung als Zahl zwischen 1 und 16 am besten, im zweiten Teil aber nicht möglich. Für Darstellungen, die die Verarbeitung ineffizient machen, können Punkte abgezogen werden.

(2) *Laufzeit des Verfahrens in Ordnung*: Für die ersten vier Beispiele müssen in kurzer Zeit Ergebnisse berechnet werden können. Liegt die Laufzeit auf dem Niveau des zweiten DP-Ansatzes oder besser, ist auch das fünfte Beispiel in kurzer Zeit lösbar; dies wurde in vielen Einsendungen erreicht. Wurde das 5. Beispiel nicht gelöst, so gibt es 2 Punkte Abzug, in noch schlechteren Fällen bis zu 4 Punkte.

(3) *Verfahren nicht unnötig ineffizient*: Umständlichkeiten, etwa bei der Implementierung, welche die Laufzeit nicht prinzipiell, aber praktisch beeinträchtigen, können hier zu (eher geringem) Punktabzug führen.

(4) *Speicherbedarf in Ordnung*: Die gegebene Hex-Zahl lässt sich problemlos direkt mit einem Speicherbedarf in $\mathcal{O}(n)$ darstellen. Der Speicherbedarf der gesamten DP-Tabelle soll $\mathcal{O}(n^2m)$ nicht deutlich überschreiten. Andersartige Verfahren sollten ebenfalls nicht mehr Speicher benötigen.

(5) *Verfahren mit korrekten Ergebnissen*: Die ausgegebene Hex-Zahl muss jeweils die größte sein, die mit maximal m Umlegungen erzeugt werden kann. Das Verfahren muss außerdem eine gültige Folge an Umlegungen berechnen können, welche die berechnete Zahl erzeugt.

(6) *Bedingungen werden eingehalten*:
- Es muss garantiert sein, dass keine Ziffer je geleert wird; dies kann unter anderem bei globalen Stapelstrukturen passieren.
- Es dürfen maximal m Umlegungen erzeugt werden.
- Die erzeugte Zahl muss eine gültige Hex-Zahl in der der Siebensegmentdarstellung sein. So ist zum Beispiel eine Ziffer ∃ nicht erlaubt.

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Es muss gut begründet werden, dass das Verfahren korrekt ist, also u. a. die größte Hex-Zahl berechnet und die dazu berechnete Folge an Umlegungen gültig ist (keine Leerung von Ziffern).

(2) *Gute Überlegungen zur Laufzeit des Verfahrens*: Die Laufzeit des Verfahrens muss nicht zwingend formal, aber nachvollziehbar und korrekt charakterisiert werden.

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Es sollten alle vorgegebenen Beispiele bearbeitet und dokumentiert worden sein. Wenn nicht alle, aber mindestens 4 der 6 Beispiele dokumentiert wurden, gibt es 2 Punkte, bei weniger bis zu 4 Punkte Abzug. Wer nicht

den effizienteren DP-Ansatz implementiert hat, kann Beispiel 5 nicht in vertretbarer Zeit lösen; das sollte dann aber dokumentiert sein. Die Ausgabe der einzelnen Umlegungen ist nur für die Beispiele 0, 1 und 2 Pflicht.

(5) *Ergebnisse nachvollziehbar dargestellt*:
- Die Hex-Zahl muss in einem geeigneten Format ausgegeben werden.
- Der Zwischenstand nach jeder Umlegung muss lesbar ausgegeben werden. In der Regel ergibt sich nicht nach jeder Umlegung eine gültige Hex-Zahl, weshalb eine graphische Ausgabe der Siebensegmentdarstellung notwendig ist.
- In der Ausgabe der Zwischenstände muss jeder Schritt einer einzelnen Umlegung, die erste (noch nicht unbedingt ausgegebene) Zahl der Hex-Zahl der Eingabe und die letzte der Lösungs-Hex-Zahl entsprechen.

Bonusaufgabe: Zara Zackigs Zurückkehr

4.1 Lösungsidee

Wir haben N Karten $v_1, \ldots, v_N$, welche sowohl die $K+1$ echten Karten von Zara enthalten als auch die $N-K-1$ zufällig hinzugefügten Karten der „Freunde“. Wir suchen nun $K+1$ Karten daraus, sodass eine Karte (die Sicherungskarte) das exklusive Oder (XOR) der anderen K Karten ist (der Schlüsselkarten). Bei einer solchen Auswahl gilt außerdem, dass jede Karte die Sicherungskarte sein könnte, da jede Karte das XOR der jeweils anderen Karten ist. Eine solche Auswahl lässt sich auch dadurch charakterisieren, dass das XOR aller dieser Karten 0 ist. Wir müssen also $K+1$ Karten finden, deren exklusives Oder 0 ist.

Naiver Ansatz

Es gibt insgesamt $\binom{N}{K+1}$ Möglichkeiten, aus den N Karten $K+1$ auszuwählen. Das sind für das Beispiel aus der Aufgabenstellung (111 Karten, mit 10 Schlüsselkarten und einer Sicherungskarte) genau 473 239 787 751 081 mögliche Kombinationen. Für jede Kombination kann das XOR berechnet und mit 0 verglichen werden. Die hohe Anzahl an Kombinationen würde jedoch zu einer inakzeptablen Laufzeit führen.

Darstellung als lineares Gleichungssystem

Jede Karte kann als ein Bitvektor mit M Bits verstanden werden. Um Zaras Problem weiter in ein Problem der linearen Algebra umzuformen, definieren wir N unbekannte Variablen $s_1, ..., s_N \in \{0,1\}$, wobei $s_i = 1$ bedeutet, dass die Karte v_i in der Lösung ist, und $s_i = 0$, dass die Karte von den „Freunden“ stammt. Eine Belegung von s_i ist dann korrekt, wenn sie das lineare Gleichungssystem

$$v_1 s_1 \oplus v_2 s_2 \oplus \cdots \oplus v_N s_N = 0,$$

wobei $\oplus$ für das exklusive Oder steht, löst und die Nebenbedingung erfüllt, dass genau $K+1$ der s_i gleich 1 sind und die anderen 0.

Betrachten wir ein Beispiel mit $N = 4$ Karten (Vektoren) mit jeweils $M = 6$ Bits (Länge der Vektoren). Gesucht sind $K+1 = 3$ Karten, darunter $K = 2$ Schlüsselkarten und eine Sicherungskarte. Die Karten werden durch die folgenden Vektoren dargestellt:

$$v_1 = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}, \; v_2 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \; v_3 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \; v_4 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Es entsteht folgendes Gleichungssystem:

$$v_1 s_1 \oplus v_2 s_2 \oplus v_3 s_3 \oplus v_4 s_4 = 0$$

Oder explizit:

$$\begin{aligned}
1s_1 \oplus 0s_2 \oplus 0s_3 \oplus 1s_4 &= 0 \\
0s_1 \oplus 1s_2 \oplus 1s_3 \oplus 1s_4 &= 0 \\
1s_1 \oplus 1s_2 \oplus 1s_3 \oplus 1s_4 &= 0 \\
1s_1 \oplus 0s_2 \oplus 0s_3 \oplus 0s_4 &= 0 \\
0s_1 \oplus 1s_2 \oplus 0s_3 \oplus 1s_4 &= 0 \\
1s_1 \oplus 0s_2 \oplus 0s_3 \oplus 1s_4 &= 0
\end{aligned}$$

Dabei sollen genau 3 der s_i gleich 1, die anderen 0 sein.

Lösung mit dem Gauß-Jordan-Verfahren

Zaras Problem lässt sich so umformuliert mithilfe des Gauß-Jordan-Verfahrens[11] und geschickten Ausprobierens der Schlüsselmengen lösen.

Notation Die Karten können als Spaltenvektoren in eine Matrix eingetragen werden. In der ersten Zeile stehen so alle Bits an der ersten Position der Karten, in der zweiten Zeile alle Bits an der zweiten Position und so weiter. Die Bits sind somit die Koeffizienten der Matrix.

Zusammen mit der Und-Operation als Multiplikation kann das lineare Gleichungssystem auch mit einer Matrix-Vektor-Multiplikation dargestellt werden. Obiges Beispiel wird dann zu:

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Es wird also ein Vektor $S = (s_1, ..., s_N)^T$ gesucht, sodass diese Gleichung erfüllt ist und genau $K+1$ Karten ausgewählt werden, also $K+1$ der s_i gleich 1 sind und die anderen gleich 0.

Gauß-Jordan-Verfahren Im Folgenden werden die Zeilen dieser Matrix mithilfe des Gauß-Jordan-Verfahrens so manipuliert, dass eine reduzierte Stufenform[12] entsteht. Hierfür wird zunächst der Nullvektor zur Matrix hinzugefügt. Dieser bleibt jedoch immer 0 und wird nur gemäß der gebräuchlichen Notation mitgeführt.

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{array}\right)$$

[11] Siehe auch https://en.wikipedia.org/wiki/Gaussian_elimination
[12] Siehe auch https://en.wikipedia.org/wiki/Row_echelon_form#Reduced_row_echelon_form

Zunächst wird bei allen Zeilen außer der ersten versucht, die erste 1 von links zu eliminieren. Hierzu wird die erste Zeile auf alle anderen Zeilen addiert, welche ebenfalls eine führende 1 haben. Zur Erinnerung: Die Addition entspricht hier dem Exklusiv-Oder-Operator $\oplus$.

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Das Gleiche passiert nun auch für die zweite, dritte und vierte Spalte. Sollte die i-te Zeile an der i-ten Spalte keine 1 haben, so wird sie mit einer Zeile vertauscht, welche dort eine 1 hat. Das Ergebnis ist dadurch weiterhin richtig, da nur für alle Karten gleichermaßen die Reihenfolge der Bits vertauscht wird und jede Karte weiterhin die eigenen Bits behält. Gibt es keine solche Zeile, wird selbiges in der nächsten Spalte versucht. Es entsteht folgende Matrix:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Hierbei sind nun Nullen unterhalb der Hauptdiagonalen entstanden. Analog wird auch für den Teil oberhalb dieser Diagonalen verfahren. Beginnend von der untersten Zeile werden nach oben Zeilen addiert, falls diese in der selben Spalte eine 1 haben wie die Zeile, welche dort eine führende 1 hat. In diesem Beispiel wird nur auf die zweite Zeile die dritte addiert. In der ersten und zweiten Zeile sind zwar ebenfalls 1en in der gleichen Spalte, allerdings ist keine von beiden eine führende 1. So entsteht folgende Matrix:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Bestimmen der Teillösungen Nachdem einige Teile der Matrix auf Nullen reduziert werden konnten, wird abgezählt, wie viele Stufen die Stufenform hat. Bei unserem Beispiel sind dies drei Stufen, hier rot markiert:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Da $K = 2$ ist, wird eine Lösung mit 3 Karten gesucht. Für alle Spalten (Karten) rechts des markierten Teils muss nun eine Auswahl getroffen werden, welche Spalten in der Lösung vorkommen und welche nicht. Für die Spalten links davon gibt es jeweils eine eindeutige Auswahl (die aus der Stufenform abgelesen werden kann), die das Gleichungssystem löst, jedoch nicht unbedingt die Nebenbedingung erfüllt. Für jede Auswahl muss also geprüft werden, ob die Nebenbedingung erfüllt ist.

In diesem Beispiel gibt es nur eine freie Spalte (die 4. Spalte), für welche die beiden Möglichkeiten (in der Lösung oder nicht) getestet werden müssen.

1. Betrachten wir zunächst den Fall, dass die 4. Karte nicht in der Lösung ist ($s_4 = 0$) und somit alle Werte der 4. Spalte mit 0 multipliziert werden. Nun steht in den ersten drei Zeilen, dass s_1, s_2 und s_3 jeweils 0 sind, wie im Gleichungssystem zu erkennen ist:

$$\begin{aligned} 1s_1 \oplus 0s_2 \oplus 0s_3 \oplus 1 \cdot 0 &= 0 \\ 0s_1 \oplus 1s_2 \oplus 0s_3 \oplus 1 \cdot 0 &= 0 \\ 0s_1 \oplus 0s_2 \oplus 1s_3 \oplus 0 \cdot 0 &= 0 \end{aligned}$$

   Also ist die Gesamtauswahl leer und die Bedingung, dass genau $K + 1 = 3$ Karten ausgewählt werden müssen, nicht erfüllt.

2. Wird jedoch angenommen, dass die 4. Karte in der Lösung ist ($s_4 = 1$), so entstehen folgende Gleichungen:

$$\begin{aligned} 1s_1 \oplus 0s_2 \oplus 0s_3 \oplus 1 \cdot 1 &= 0 \\ 0s_1 \oplus 1s_2 \oplus 0s_3 \oplus 1 \cdot 1 &= 0 \\ 0s_1 \oplus 0s_2 \oplus 1s_3 \oplus 0 \cdot 1 &= 0 \end{aligned}$$

   Die konstanten Terme werden auf die rechte Seite gebracht:

$$\begin{aligned} 1s_1 \oplus 0s_2 \oplus 0s_3 &= 1 \\ 0s_1 \oplus 1s_2 \oplus 0s_3 &= 1 \\ 0s_1 \oplus 0s_2 \oplus 1s_3 &= 0 \end{aligned}$$

   Es gilt also $s_1 = s_2 = 1$ und somit sind die Karten v_1 und v_2 in der Lösung, zusammen mit unserer Vorauswahl v_4. In der Tat erfüllt dies die Nebenbedingung, dass die Auswahl aus drei Karten bestehen muss. Die Lösung lautet also v_1, v_2, v_4.

Sofern es Spalten rechts der Stufenform gibt (bei den BWINF-Beispielen in der Regel $N - M$ Karten, bzw. 0, wenn $N < M$), muss jede Auswahl daraus auf Gültigkeit getestet werden, indem gezählt wird, wie viele der Einträge den Wert 1 haben. Für die gebundenen Variablen lassen sich wie oben bereits erwähnt aus der Stufenformel explizite Terme in Abhängigkeit der freien Variablen ablesen, in die jeweils nur eingesetzt werden muss.

Es müssen natürlich nur Auswahlen der freien Variablen geprüft werden, bei denen höchstens $K + 1$ der Einträge 1 sind.

Beim Bestimmen der Lösungen bietet es sich an, die Anzahl der gewählten freien Spalten erst zu erhöhen, nachdem alle Möglichkeiten mit einer Anzahl getestet wurden. Mit anderen Worten: Zuerst testet man alle Kombinationen mit null Spalten, dann alle Kombinationen mit einer Spalte, dann mit zwei Spalten und so weiter. Dies hängt damit zusammen, dass die Wahrscheinlichkeit gering ist, dass alle $K + 1$ gesuchten Karten in den letzten $N - M$ Spalten vorkommen,

wenn wir davon ausgehen, dass die „Freunde“ gut gemischt haben und die Verteilung der Karten gleichverteilt zufällig ist.

Eindeutigkeit

Um zu testen, ob die gefundene Lösung die einzige Lösung ist, könnten alle Möglichkeiten getestet werden. Da jedoch in der Aufgabenstellung beschrieben wurde, dass die hineingemischten Karten zufällig generiert wurden, ist die Wahrscheinlichkeit für weitere Lösungen sehr gering: Die Exklusiv-Oder-Summe jeder der $\binom{N}{K+1} - 1$ nicht richtigen Teilmengen ist unabhängig gleichverteilt zufällig (unter den 2^M Möglichkeiten), und daher ist die Wahrscheinlichkeit einer zufälligen Lösung:

$$P(N,M,K) = 1 - \left(1 - \frac{1}{2^M}\right)^{\binom{N}{K+1}-1}.$$

Für die Beispiele auf der BWINF-Website ist diese Abschätzung immer kleiner als 10^{-4}.

Implementierung

Für die Implementierung eignen sich Bit-Arrays. Ein *Bit-Array* ist ein Array von Bits, das auf einer Maschine mit 64-Bit-Integern als (kürzeres) Array von Integern dargestellt werden kann, deren 64 Bits jeweils bis zu 64 Bits des Bit-Arrays entsprechen.

Nachdem die Karten eingelesen wurden, werden Bit-Arrays verwendet, um die Zeilen der Matrix darzustellen, und die Werte der Karten als Spaltenvektoren in die Matrix eingetragen. Darauf folgt das Gauß-Jordan-Verfahren, angewendet auf die Zeilen. Die dabei nötigen Zeilenoperationen (hier: Additionen von Zeilen) werden, da es sich um Operationen von Bit-Arrays handelt, um einen konstanten Faktor, etwa 64, schneller durchgeführt.

Nachdem die Anzahl der Stufen festgestellt wurde, werden die noch freien Spaltenvektoren wiederum in Bit-Arrays umgewandelt und daraufhin mit diesen Arrays das Bestimmen der restlichen Schlüssel durchgeführt. Auch die dafür nötigen Berechnungen werden dadurch um einen konstanten Faktor schneller.

Hierfür lohnt es sich, eine schnelle Methode zu entwickeln, die Anzahl der Einsen in einem Integer zu zählen. Hierfür kann neben der naiven Methode auch Brian Kernighans Algorithmus[13], Bitmanipulation[14] oder die Assembly-Instruktion popcnt[15] verwendet werden.

Laufzeitanalyse

Bei N verschiedenen Karten der Länge M das Gauß-Jordan-Verfahren anzuwenden, hat einen Aufwand von

$$T = \mathcal{O}(NM^2),$$

da wir jede der M Zeilen auf höchstens M verschiedene Zeilen addieren müssen und diese Operation für je N Bits ausführen. Die Laufzeit der Bestimmung der Teillösungen ist abhängig davon, wie viele Stufen bei dem Gauß-Jordan-Verfahren entstanden sind. Gilt $N < M$, gibt es

[13] Siehe https://graphics.stanford.edu/ seander/bithacks.html#CountBitsSetKernighan
[14] Siehe http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel
[15] Siehe https://www.felixcloutier.com/x86/popcnt

also weniger Karten, als jede Karte Bits hat, wie bei dem Beispiel aus der Aufgabenstellung, so werden meist N Stufen entstehen und der Teil, der im Weiteren betrachtet werden muss, ist gering (bzw. nicht vorhanden). Sollte jedoch $N > M$ gelten, es also mehr Karten geben, als jede Karte Bits hat, wie bei `stapel3.txt`, `stapel4.txt` und `stapel5.txt`, so muss bei mindestens $N - M$ Spalten geprüft werden, ob sie in der Lösung sind oder nicht. Da einige Teillösungen direkt ausgeschlossen werden, und wenn wir davon ausgehen, dass $N-M$ Spalten rechts von der Stufenform übrig bleiben, so müssen trotzdem noch bis zu

$$\sum_{i=0}^{K+1} \binom{N-M}{i}$$

Auswahlen an Spalten überprüft werden und für jede davon die restlichen Variablen s_i berechnet werden. Im Erwartungswert enthält die richtige Auswahl jedoch nur $\frac{K+1}{N}(N-M)$ der $N-M$ Karten, sodass die tatsächliche Anzahl an überprüften Auswahlen deutlich geringer ist.

4.2 Auswahl des korrekten Schlüssels

Nachdem Zara ihre Schlüsselkarten $v_{j_1}, v_{j_2}, \ldots, v_{j_{K+1}}$ gefunden hat, kann sie diese erneut aufsteigend sortieren (wir nehmen an, dass dies bereits die Reihenfolge nach Sortierung ist). Möchte sie nun die i-te Tür öffnen, kann sie zunächst die Karte v_{j_i} testen. Sollte diese Karte nicht die Tür öffnen, kann dies nur daran liegen, dass der Sicherungsschlüssel vor dem gesuchten Schlüssel eingeordnet wurde. Also wurde der gesuchte Schlüssel um eins verschoben, woraus folgt, dass die Karte $v_{j_{i+1}}$ die richtige ist.

Zara wird so tatsächlich nicht mehr als *einen* Fehlversuch benötigen, um das Haus aufzusperren.

4.3 Weiterer Lösungsansatz (Teile und Herrsche)

Das Problem lässt sich auch mit dem folgenden Teile-und-Herrsche-Algorithmus[16] angehen.

Zunächst speichert man für alle Teilmengen der ersten $\lfloor (K+1)/2 \rfloor$ Karten ihr Exklusiv-Oder zusammen mit der Teilmenge in einer nach dem Exklusiv-Oder sortierten Datenstruktur. Diese kann beispielsweise als Hashtabelle realisiert werden. Daraufhin wird für alle Teilmengen der restlichen $\lceil (K+1)/2 \rceil$ Karten getestet, ob ihr Exklusiv-Oder in der Datenstruktur ist oder nicht. Ein Treffer entspricht einer Lösung, nämlich der Vereinigung der beiden Teilmengen von Karten. Dieser Algorithmus hat einen Zeitaufwand von

$$T = \mathcal{O}\left(M \cdot \binom{N}{\lceil (K+1)/2 \rceil} \cdot \log \lfloor (K+1)/2 \rfloor\right).$$

Der logarithmische Faktor ist für das Suchen von Elementen in der Datenstruktur nötig. Durch diesen Trick der Aufteilung der Karten in zwei Teile ergibt sich also eine deutlich bessere Laufzeit im Vergleich zu der der naiven Lösung von $\mathcal{O}\left(M \cdot \binom{N}{K+1}\right)$.

Mit diesem Algorithmus lassen sich alle Beispiele außer `stapel3.txt` und `stapel4.txt` in unter einer Stunde lösen. Wirklich geeignet ist er aber vor allem bei `stapel5.txt`, worauf er,

[16] Siehe auch https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm

da $N - M = 136$ sehr groß, aber dafür $K = 4$ sehr klein ist, eine deutlich bessere Laufzeit als der das Gauß-Jordan-Verfahren benutzende Lösungsansatz hat. Bei dieser Aufgabe sind also grundsätzlich verschiedene Ansätze für in verschiedenen Dimensionen große Eingaben jeweils besser oder schlechter geeignet. Insbesondere kann nicht pauschal davon gesprochen werden, dass einer dieser beiden Ansätze die *bessere* Laufzeit habe.

4.4 Beispiele

Im folgenden Abschnitt werden die (mit sehr hoher Wahrscheinlichkeit eindeutigen) Lösungen für die von BWINF vorgegebenen Beispiele vorgestellt. Bei jeder Lösung wird eine Tabelle angegeben mit den Indizes (beginnend mit 0, nach der Reihenfolge in der Datei) der Karten zusammen mit den ersten 32 Bits der Codewörter. Die Karten sind für Zara aufsteigend sortiert.

stapel0.txt

$$N = 20, M = 32, K = 4$$

$$\text{Laufzeit} < 1\,\text{ms}$$

Index				
2	00111101	01011100	01101001	10011001
9	10101100	11111101	10101000	11100000
11	10111000	01100111	00001010	10111110
5	11010111	11101011	11011011	11110000
3	11111110	00101101	00010000	00110111

stapel1.txt

$$N = 20, M = 32, K = 8$$

$$\text{Laufzeit} < 1\,\text{ms}$$

Index				
15	00010001	11010011	00011111	01100100
1	00100000	11110011	11101111	01111100
11	00100011	10011101	10101110	11100011
4	00110100	00101010	01000011	11010010
7	00110110	00011010	11010111	11111010
14	11000111	11101011	01000001	01110100
2	11010011	01011011	01010011	01010111
6	11110011	10101100	10010000	10111110
9	11110111	10010001	01001000	01001110

stapel2.txt

$$N = 111, M = 128, K = 10$$

$$\text{Laufzeit} \approx 1\,\text{ms}$$

Index					
57	00101000	01100001	00101110	11101011	...
20	00101011	11100010	10110101	10111100	...
89	01101001	00101100	01010011	11111101	...
9	01101011	10100011	01110100	01100001	...
95	01110110	01111000	11100111	10001101	...
53	10000000	00010010	01100110	01000110	...
26	10101011	00000110	11000001	01111111	...
76	10101111	11001001	00101001	11101100	...
71	11000011	00010011	01110001	01100100	...
78	11011110	00010100	11011111	00110000	...
99	11101110	10101110	01111011	11000111	...

stapel3.txt

$$N = 161, M = 128, K = 10$$

Laufzeit $\approx 2\,\text{ms}$

Index					
133	00100000	11100111	00011010	10001111	...
43	01010000	10110111	00111100	01110011	...
120	01110001	11111010	10000100	11100011	...
127	01110110	10011100	10000110	11100100	...
98	01111101	10001010	11001100	11101011	...
71	10110000	00100110	01101101	01000100	...
23	10110111	01001011	11101100	11000101	...
26	10111000	10111110	00101111	10101010	...
110	10111111	01010100	11000001	01101100	...
144	11000001	01100100	01101001	11011111	...
34	11001011	01011111	11101110	10001000	...

stapel4.txt

$$N = 181, M = 128, K = 10$$

Laufzeit $\approx 8\,\text{ms}$

Index					
76	00000111	01101001	01011011	10001111	...
23	00101100	00011110	11110000	10000001	...
91	00101100	00111000	11100111	10000100	...
83	00110110	01011100	10010011	11001111	...
95	01000011	00100110	10110011	11011101	...
116	10000001	00010111	00110101	00011000	...
159	10001100	00101100	11010010	11000110	...
154	10101010	00001111	11110011	11011110	...
146	11000101	11000100	10000010	11101000	...
0	11100010	00000011	11010011	11111001	...
160	11110010	11000110	00101001	10001001	...

stapel5.txt

$$N = 200, M = 64, K = 4$$

$$\text{Laufzeit} \approx 865\,\text{ms}$$

Index					
167	01011111	11000111	00000010	11111000	...
70	10000100	11101010	00111110	01001101	...
185	10100001	10101100	10111011	10011000	...
163	10101110	11001100	10011000	11001100	...
77	11010100	01001101	00011111	11100001	...

4.5 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Die Karten müssen vollständig verwendet und das bitweise Exklusiv-Oder korrekt berechnet werden. Zudem muss in Betracht gezogen werden, dass es mehrere Lösungen geben kann.

(2) *Laufzeit des Verfahrens in Ordnung*: Jeder Algorithmus mit einer Laufzeit von $\mathcal{O}\left(\binom{N}{K+1}\right)$ oder schlechter ist als ineffizient zu betrachten und mit Punktabzug zu bewerten. Algorithmen, welche die Laufzeit stark genug einschränken, sollten hier keinen Abzug bekommen.
Praktischer gesprochen: Es gibt viele Verfahren, welche die kleineren Beispiele lösen können. Sollte ein Verfahren alle Beispiele lösen, ist es als sehr gut einzuschätzen. Falls ein Verfahren das Beispiel aus der Aufgabenstellung löst (und wohl auch die kleineren Beispiele), gibt es keine Punktabzüge. Falls auch die herausfordernden Beispiele (`stapel3.txt` bis `stapel5.txt`) gelöst werden, so können bis zu zwei Bonuspunkte vergeben werden: 1 Punkt für mindestens zwei dieser Beispiele, 2 Punkte für alle drei.
Ein besonderer Fall ist der geschilderte Ansatz nach dem Prinzip „teile und herrsche“. Wird er allein verwendet, können die Beispiele 3 und 4 nicht gelöst werden und Beispiel 2 nur mit Mühe; dann werden Punkte abgezogen. Gleichzeitig ist er aber wichtig für Beispiel 5, so dass es ohne ihn möglicherweise nicht die vollen Bonuspunkte geben kann.

(3) *Verfahren nicht unnötig ineffizient*: Es sollte eine Datenstruktur verwendet werden, die für den gewählten Algorithmus geeignet ist. So ist zum Beispiel die Verwendung von Bitarrays oder Bitvektoren notwendig für eine effiziente Implementierung, wenn diese auf dem Gauß-Jordan-Verfahren basiert.

(4) *Verfahren mit korrekten Ergebnissen*: Das Ergebnis ist immer eine Auswahl an $K+1$ Karten, die alle in der Ursprungsmenge vorkommen. Die Exklusiv-Oder-Summe über die finale Auswahl der Schlüssel sollte den Nullvektor ergeben.

(5) *Verfahren funktioniert uneingeschränkt*: Das Verfahren kann Problemstellungen für alle möglichen N, M und K bearbeiten. Es dürfen keine Bedingungen an die Eingabeparameter gesetzt weden, wie zum Beispiel, dass N kleiner als M ist.

(6) *Verfahren zur Schlüsselauswahl angegeben*: Auch Aufgabenteil b) muss beantwortet werden. Dazu gehört die Angabe eines Verfahrens zur Auswahl des korrekten Schlüssels aus den $K+1$ gefundenen Schlüssel- und Sicherungskarten.

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Das Verfahren und insbesondere seine Korrektheit müssen gut nachvollziehbar begründet werden. Bei der Verwendung bereits bekannter Algorithmen müssen diese erklärt oder referenziert werden; aus der Dokumentation muss klar werden, dass verstanden wurde, was die „fremden“ Verfahren tun und warum sie zur Lösung nützlich sind.

(2) *Gute Überlegungen zur Laufzeit des Verfahrens*: Die Laufzeit des Verfahrens lässt sich recht einfach erkennen und sollte auch korrekt in der Dokumentation vorkommen. Die Laufzeit muss nicht zwingend formal, aber nachvollziehbar und korrekt charakterisiert werden.

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Alle Beispiele der BWINF-Website sollten dokumentiert und nachvollziehbar sein. Sollte ein Beispiel nicht gelöst werden können, so sollte dies erwähnt werden.

(5) *Ergebnisse nachvollziehbar dargestellt*: Die Ergebnisse zu den Beispielen sollten die Bits der Schlüsselkarten und der Sicherungskarte jeweils in einer Zeile gelistet haben. Es muss entweder der ursprüngliche Index der ausgewählten Karten mit angegeben werden oder diese zumindest sortiert abgebildet werden.

Aus den Einsendungen: Perlen der Informatik

Aufgabe 1: Müllabfuhr

Verbesserungen -> Homeoffice einführen: Eine Funktionalität [...] ist die Möglichkeit in der Zentrale zu bleiben.

Aufgabe 2: Rechenrätsel

Es würde beim Ausprobieren von 100 Millionen Kombinationen pro Sekunde circa $1.7 \cdot 10^{284}$ Jahre dauern, um alle Kombinationen auszuprobieren. Das ist länger als die „Al dente“ Kochzeit von Hartweizennudeln (in Wasser).

Die Durchführung einer Laufzeitanalyse ergibt bei diesem Programm keinen Sinn, weil die Laufzeit viel mehr vom Zufall abhängt, als von den geforderten Operationen.

Dadurch wird eine Ziffer mit einer höheren Wertigkeit gezwungen, sich zu erniedrigen.

Die theoretische Laufzeitanalyse hierfür ist schwer und sinnlos.

Aufgabe 3: Hex-Max

Funktionsname: `berchne_die_maximalen_werte_die_nach_jeder_ziffer_dazu_und_weg_gelegt_werden_koennen()`

Der Test bestätigt, dass das Programm wahrscheinlich funktioniert. Schließlich ist die Ausgabe nicht kleiner als die Eingabe.

Auf diese Weise entsteht eine Art „Mini-KI“, die sich sowohl Sachen merken kann als auch selber Schlüsse ziehen kann.

die Methode `give_to_add_to_remove()`

Die Laufzeit liegt zwischen linear und exponentiell (inklusive Schaubild, dass dies graphisch verdeutlicht).

Die Laufzeit meines Programmes beträgt O(n). Dies gilt aber nicht für einen bestimmten Sonderfall.

Das Stäbchen mittig in der Mitte

`sys.setrecursionlimit(len(ziffern) * len(versuchsliste) + 1)` $\sim$ Pictures taken seconds before disaster

Ausgleich von Angebot und Nachfrage - Tabelle: Die Ziffernsysteme können auch als Markt betrachtet werden.

Für die Lösung der restlichen Beispiele ist mein Programm leider noch nicht optimiert genug und würde vermutlich bis zur Abgabe der 2. Runde nächstes Jahr rechnen.

Für die horizontalen Striche nutze ich das Ascii Zeichen 196 und für die vertikalen Striche 179.

Bonusaufgabe: Zara Zackigs Zurückkehr

Alle Beispiele werden vom Gefühl her direkt gelöst.

Hierbei eine Anmerkung zu Beginn: es handelt sich nur um Überlegungen, die nicht tiefgründig durchdacht worden sind.

...die geplottet mit anderen eine ungefähre Gerade -> lineare Zeit erreicht. O(n) wäre also im schlimmsten Fall O(n).

Um auch die größten Beispiel durchlaufen lassen zu können, wird Parallelität benötigt um den gegebenen Arbeitsspeicher nicht zu crashen.