43. Bundeswettbewerb Informatik 2024/2025, 1. Runde

Lösungshinweise

Allgemeines

Es ist sehr erfreulich, dass wieder sehr viele die Aufgaben bearbeitet und am Bundeswettbewerb Informatik teilgenommen haben. Den Veranstaltern ist bewusst, dass in der Regel viel Arbeit hinter der Erstellung einer Einsendung steckt.

Bei der Bewertung wurden die in den Einsendungen gezeigten Leistungen so gut wie möglich gewürdigt. Das ist nicht immer leicht, insbesondere wenn die Dokumentation nicht die im Aufgabenblatt genannten Anforderungen erfüllt. Bevor mögliche Lösungsideen zu den Aufgaben und Einzelheiten zur Bewertung beschrieben werden, soll deshalb im folgenden Abschnitt auf die Dokumentation näher eingegangen werden; vielleicht helfen diese Anmerkungen bei der nächsten Teilnahme.

Wie auch immer die Bewertung einer Einsendung ausfällt, sie sollte nicht entmutigen! Allein durch die Arbeit an den Aufgaben und ihren Lösungen hat jede Teilnehmerin und jeder Teilnehmer einiges gelernt; diesen Effekt sollte man nicht unterschätzen.

Die Bearbeitungszeit für die 1. Runde beträgt etwa drei Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der Dokumentation zu vermeiden. Aufgaben sind gelegentlich schwerer zu bearbeiten, als es auf den ersten Blick erscheinen mag. Erst bei der konkreten Umsetzung einer Lösungsidee oder beim Testen von Beispielen kann man auf Besonderheiten oder Schwierigkeiten stoßen, die zusätzlicher Zeit bedürfen.

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

Dokumentation

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

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

Ganz besonders wichtig sind die vorgegebenen (und ggf. weitere) *Beispiele*. Wenn Beispiele und die zugehörigen Ergebnisse in der Dokumentation ganz oder teilweise fehlen, führt das zu Punktabzug. Zur Bewertung ist für jede Aufgabe vorgegeben, zu wie vielen (und teils auch zu welchen) Beispielen Programmausgaben oder Ergebnisse in der Dokumentation erwartet werden. Diese Ergebnisse sollten idealerweise korrekt sein. Punktabzug für falsche Ergebnisse gibt es aber nur, wenn nicht schon für den ursächlichen Mangel ein Punkt abgezogen wurde.

Es ist nicht ausreichend, Beispiele nur in gesonderten Dateien abzugeben, ins Programm einzubauen oder den Bewerterinnen und Bewertern sogar das Erfinden und Testen geeigneter Beispiele zu überlassen. Beispiele sollen die Korrektheit der Lösung belegen und auch zeigen, wie das Programm mit Sonderfällen umgeht.

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

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

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

Bewertung

Bei den im Folgenden beschriebenen Lösungsideen handelt es sich um Vorschläge, aber sicher nicht um die einzigen Lösungswege, die korrekt sind. Alle Ansätze, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind, werden in der Regel akzeptiert. Einige

Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall erwartet werden. Zu jeder Aufgabe werden deshalb im jeweils letzten Abschnitt die Kriterien näher erläutert, auf die bei der Bewertung dieser Aufgabe besonders geachtet wurde. Die Kriterien sind in der Bewertung, die man im AMS einsehen kann, aufgelistet. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation, die oben erklärt sind.

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

Für die Gesamtpunktzahl sind die drei am besten bewerteten Aufgabenbearbeitungen maßgeblich. Es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreicht man mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten und einen 3. Preis mit 9 bis 11 Punkten. Mit einem 1. oder 2. Preis ist man für die 2. Runde qualifiziert. Kritische Fälle mit nur 11 Punkten sind bereits sehr gründlich und mit viel Wohlwollen geprüft. Leider ließ sich aber nicht verhindern, dass Teilnehmende nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden; dies ist ziemlich riskant, da sich leicht Fehler einschleichen.

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

Danksagung

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

An der Erstellung der im Folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Philip Gilde (Junioraufgabe 1), Christina Suttrop (Junioraufgabe 2), Christina Suttrop (Aufgabe 1), Tobias Steinbrecher (Aufgabe 2), Martin Bartram (Aufgabe 3), Raphael Gaedtke (Aufgabe 4) und Tim Pokart (Aufgabe 5). Allen Beteiligten sei für ihre Mitarbeit ganz herzlich gedankt.

Junioraufgabe 1: Quadratisch, praktisch, grün

J1.1 Lösungsidee

So quadratisch wie möglich

Um eine Lösung mit möglichst quadratischen Gärten zu finden, müssen wir uns überlegen, wann ein Garten „quadratischer“ als ein anderer ist. Ein Garten ist immer dann quadratisch, wenn er genau so hoch wie breit ist. Solche Gärten sind immer quadratischer als solche, bei denen das nicht der Fall ist. Um zwei Gärten zu vergleichen, die nicht quadratisch sind, können wir vergleichen, wie ähnlich die Höhe und Breite sind, so dass Gärten mit ähnlicheren Seitenlängen quadratischer sind als solche mit unterschiedlicheren Seitenlängen. Dabei wollen wir auch, dass bestimmte Gärten gleich quadratisch sind. Einerseits sollen Gärten mit gleichen Proportionen, aber unterschiedlicher Größe, nicht unterschiedlich quadratisch sein. Andererseits soll ein Garten genau so quadratisch wie ein anderer sein, wenn er so breit ist wie der andere hoch und so hoch wie der andere breit.

Mit diesen drei Bedingungen können wir ein Maß bzw. eine Kostenfunktion formulieren, die höher ist, wenn ein Garten weniger quadratisch ist, und niedriger, wenn er quadratischer ist. Um zwei Gärten zu vergleichen, müssen wir dann nur noch die Werte der Kostenfunktion vergleichen. Die Kostenfunktion f ordnet also einer Höhe h und einer Breite b des Gartens die Kosten, also die Unquadratischkeit, $f(h,b)$ zu.

Ein erster Ansatz für die Kostenfunktion ist das Verhältnis von Höhe und Breite, das heißt $f(h,b) = \frac{h}{b}$. Ein quadratischer Garten hat damit die Kosten 1 und Gärten, die höher als breit sind, haben immer höhere Kosten je unterschiedlicher Höhe und Breite sind. Diese Kostenfunktion bleibt auch für gleiche Proportionen gleich, wenn sich die Fläche des Gartens ändert. Leider wird sie kleiner, wenn Gärten breiter als hoch sind, und bleibt nicht gleich, wenn man Breite und Höhe vertauscht. Um dieses Problem zu lösen können wir entweder immer das Verhältnis von der längeren zur kürzeren Seite messen, das heißt $f(h,b) = \frac{\max(h,b)}{\min(h,b)}$ oder wir addieren den Kehrwert des Verhältnisses, also $f(h,b) = \frac{h}{b} + \frac{b}{h}$. Da die erste Funktion nicht überall ableitbar ist, entscheiden wir uns für die zweite.

Die richtige Aufteilung finden

Der in diesem Abschnitt beschriebene Lösungsansatz reicht aus, um die Aufgabe vollständig zu lösen. Ein Ansatz in diesem Rahmen wird von den Teilnehmenden erwartet. Wenn dich interessiert, welche weitere Analyse und Ausarbeitung möglich wäre, lies den Abschnitt **„Exkurs zur Junioraufgabe 1: Quadratisch, praktisch, grün“** am Ende dieses Dokuments vor den Perlen.

Wir müssen nun eine Aufteilung des Grundstückes in Gärten finden, die die Kostenfunktion minimiert, ohne die Grenzen für die Anzahl an Grundstücken zu verletzen. Dafür eignet sich der Brute-Force-Ansatz, bei dem wir einfach alle möglichen Aufteilungen durchprobieren und die beste als Lösung nehmen.

Da nur gleich große rechteckige Gärten erlaubt sind, müssen wir nur alle erlaubten Gitter durchgehen. Damit wir nicht weniger Gärten als Interessenten und nicht mehr als 10% mehr Gärten als Interessenten haben, suchen wir also alle Lösungen für $k \leq nm \leq 1.1k$ mit $n,m \in \mathbb{N}^+$ (also den natürlichen Zahlen ungleich Null), wobei k die Anzahl an Interessenten, n die Anzahl an

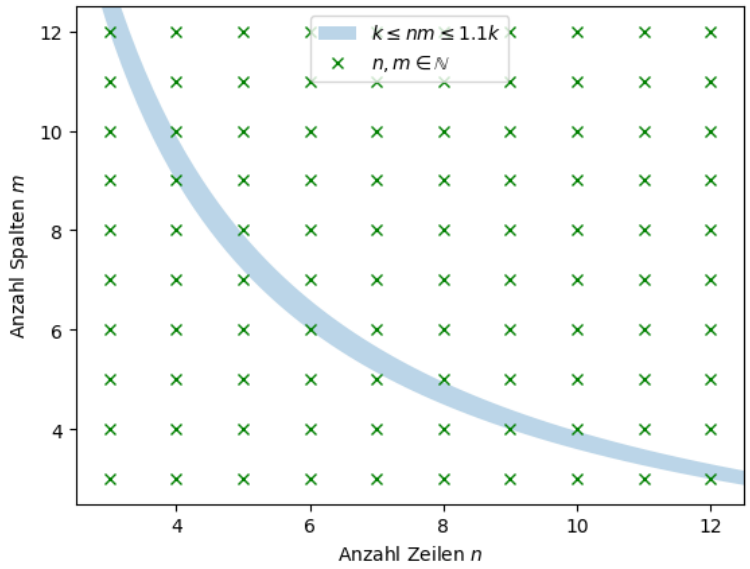


Abbildung J1.1: Im blauen Bereich erfüllen die Aufteilungen die Ungleichung $k \leq nm \leq 1.1k$. Bei den grünen Kreuzen sind Spalten- und Zeilenzahl natürliche Zahlen, so dass erlaubte Lösungen grüne Kreuze im blauen Bereich sind.

Zeilen und m die Anzahl an Spalten im Gitter ist. Diese Bedingungen sind in Abbildung J1.1 zu sehen.

Da $1 \leq n$ und $1 \leq m$, ist $n \leq 1.1k \leftrightarrow n \leq \lfloor 1.1k \rfloor$ und $m \leq 1.1k \leftrightarrow m \leq \lfloor 1.1k \rfloor$, womit wir eine erste Brute-Force-Lösung haben: Wir suchen alle Paare von $n, m \in \mathbb{N}$ mit $1 \leq n \leq \lfloor 1.1k \rfloor$ und $1 \leq m \leq \lfloor 1.1k \rfloor$, und falls $k \leq nm \leq 1.1k$ gilt vergleichen wir die Kosten dieser Lösung mit der bisher besten. Daraus ergibt sich Algorithmus 1 ($\lfloor x \rfloor$ bedeutet x abrunden und $\lceil x \rceil$ bedeutet x aufrunden).

Algorithmus 1 Brute-Force-Verfahren

procedure BRUTE FORCE(Höhe H, Breite B, Interessenten k)
  $n^*, m^* \leftarrow 0$
  $f^* \leftarrow \infty$ ▷ n^*, m^* und $f*$ sind die bisher beste Zeilen- und Spaltenzahl und Kosten.
  for $n \in 1, \ldots, \lfloor 1.1k \rfloor$ **do**
    for $m \in 1, \ldots, \lfloor 1.1k \rfloor$ **do**
      if $k \leq nm \leq 1.1k \wedge f(\frac{H}{n}, \frac{B}{m}) < f^*$ **then**
        $n^* \leftarrow n$
        $m^* \leftarrow m$
        $f^* \leftarrow f(\frac{H}{n}, \frac{B}{m})$
      end if
    end for
  end for
  return n^*, m^*
end procedure

Umgang mit Kommazahlen

In dieser Aufgabe wird an verschiedenen Stellen mit Kommazahlen gearbeitet, die verglichen und ausgegeben werden. Computer stellen Kommazahlen meistens mit begrenzter Genauigkeit[1] dar, wodurch Fehler auftreten können. Da solche Zahlen in den meisten Programmiersprachen auf ungefähr 16 Ziffern genau sind, spielt das für die Berechnungen in dieser Aufgabe eine untergeordnete Rolle, denn die Größenordnungen der Beispieleingaben sind sehr viel kleiner. Fehler können dennoch auftreten, wenn wir Kommazahlen auf Gleichheit prüfen, denn Zahlen, die eigentlich gleich sein sollten, können sich durch Rundungsfehler ab ungefähr der 16. Stelle unterscheiden. Allerdings prüfen wir in dieser Lösung nur, ob eine Kommazahl (die Kosten) kleiner als eine andere ist. So könnte das Ergebnis des hier entwickelten Verfahrens in Wahrheit nicht die quadratischste sein, sie würde sich in der Quadratischkeit aber von der besten erst ab der 16. Stelle unterscheiden, was vernachlässigbar ist.

Auch die Ergebnisse können Kommazahlen sein: Im Beispiel **garten2.txt** hat ein optimaler Garten beispielsweise eine periodische Breite und Höhe. Diese können wir so nicht ausgeben, sondern müssen sie irgendwie runden. Dabei müssen wir uns einerseits für die Anzahl an Nachkommastellen, auf die wir runden wollen, entscheiden, und andererseits für die Art des Rundens, also Aufrunden, Abrunden, kaufmännisch Runden oder auf irgendeine andere Weise. Wenn wir aufrunden, werden alle Gärten ein wenig zu groß, weshalb wir ein etwas größeres Grundstück bräuchten, um alle Gärten gleich groß zu machen. Wenn wir hingegen abrunden, bleibt am Ende des Grundstücks ein schmaler Streifen frei. Da wir hier kein größeres Grundstück brauchen, bevorzugen wir diese Rundungsweise. Wenn wir kaufmännisch runden, also abrunden, wenn die nächste Nachkommastelle kleiner als 5 und sonst aufrunden, wird dieses Verhalten unvorhersehbar, weshalb wir diese Rundungsweise vermeiden. Die Lösungen der Beispiele sind auf vier Nachkommastellen, also $0,1$ mm, genau angegeben. So genau werden Grundstücke in der echten Welt ohnehin nicht festgelegt, weshalb eine genauere Ausgabe nicht notwendig ist.

Laufzeit

Überlegungen zur Laufzeit eines Algorithmus werden in der ersten Runde nicht von den Teilnehmenden erwartet. Dennoch ist es nützlich zu verstehen, wie sich ein Algorithmus in Abhängigkeit zu den eingegebenen Größen verhält. Außerdem werden solche Überlegungen in der zweiten Runde des Bundeswettbewerbs Informatik erwartet.

Der beschriebene Brute-Force-Ansatz probiert alle Paare von natürlichen Zahlen zwischen 1 und $1.1k$ durch und führt eine konstante Anzahl an Rechenschritten durch. Davon gibt es $(\lfloor 1.1k \rfloor)^2$, das heißt der Algorithmus hat eine Laufzeit von $\mathcal{O}(k^2)$, denn die Abrundung verändert den Wert um nicht mehr als 1, und lineare Koeffizienten sind für die Big-O-Notation irrelevant.

J1.2 Beispiele

garten0.txt
4 Zeilen, 6 Spalten, 24 Gärten, ein Garten ist 10.5 m hoch und 11 m breit. Verhältnis längerer

[1] `https://de.wikipedia.org/wiki/Gleitkommazahl`

zu kürzerer Seite: 1.048.

garten1.txt
5 Zeilen, 4 Spalten, 20 Gärten, ein Garten ist 3 m hoch und 3 m breit. Verhältnis längerer zu kürzerer Seite: 1.

garten2.txt
6 Zeilen, 6 Spalten, 36 Gärten, ein Garten ist 9.1666 m hoch und 12.8333 m breit. Verhältnis längerer zu kürzerer Seite: 1.4.

garten3.txt
10 Zeilen, 11 Spalten, 110 Gärten, ein Garten ist 1.5 m hoch und 1.3636 m breit. Verhältnis längerer zu kürzerer Seite: 1.1.

garten4.txt
5 Zeilen, 264 Spalten, 1320 Gärten, ein Garten ist 7.4 m hoch und 7.5757 m breit. Verhältnis längerer zu kürzerer Seite: 1.023.

garten5.txt
120 Zeilen, 308 Spalten, 36960 Gärten, ein Garten ist 3.0416 m hoch und 3.0422 m breit. Verhältnis längerer zu kürzerer Seite: 1.

J1.3 Bewertungskriterien

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

- [−1] **Modellierung ungeeignet**
  Ein geeignetes Maß für „möglichst quadratisch“ muss einen Vergleich beliebiger Paare von Gärten erlauben.
- [−1] **Lösungsverfahren fehlerhaft**
  Die Gärten müssen der Definition entsprechend möglichst quadratisch sein. Das Verhältnis von längerer zu kürzerer Seite einer Gartens darf nicht mehr als 1% über dem in den Lösungshinweisen liegen. Es dürfen nicht weniger Gärten entstehen als es Interessenten gibt und maximal 10% mehr.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Es ist sinnvoll, die Höhe und Breite der Gärten sowie die Gesamtanzahl der Gärten anzugeben. Die Angabe einer Aufteilung des Grundstücks in $i \times j$ Gärten ist akzeptabel, wenn klar wird, welcher der Werte i und j sich auf die Höhe und welcher sich auf die Breite des Grundstücks bezieht.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens vier der sechs vorgegebenen Beispiele (`garten0.txt` bis `garten5.txt`) enthalten.

		Interessenten	
Bsp.	|Grundstück| [m^2]	Orig.	Max.
0	$42 \times 66 = 2772$	23	25
1	$15 \times 12 = 180$	19	20
2	$55 \times 77 = 4235$	36	39
3	$15 \times 15 = 225$	101	111
4	$37 \times 2000 = 74000$	1200	1320
5	$365 \times 937 = 342005$	35000	38.500

	Gärten		**Quotient**		**Differenz**
Bsp.	#	h × b [m]	$\frac{\max(h,b)}{\min(h,b)}$	+1%	$\lvert(h-b)\rvert$
0	$4 \times 6 = 24$	$10,5 \times 11$	1,048	1,058	0,5
1	$5 \times 4 = 20$	3×3	1	1,01	0
2	$6 \times 6 = 36$	$9,1666 \times 12,8333$	1,4	1,414	3,6664
3	$10 \times 11 = 110$	$1,5 \times 1,3636$	1,1	1,111	0,1364
4	$5 \times 264 = 1320$	$7,4 \times 7,5757$	1,023	1,033	0,1757
5	$120 \times 308 = 36960$	$3,0416 \times 3,0422$	1	1,01	0,0006

Junioraufgabe 2: Texthopsen

J2.1 Lösungsidee

In dieser Aufgabe soll für einen Text bestimmt werden, ob Amira oder Bela beim Texthopsen im gegebenen Text gewinnt. Das grundlegende Verfahren des Texthopsens kann aus der Aufgabe übernommen werden. So nehmen wir einen vorgegebenen Text und lassen die beiden darin hopsen: Bela fängt beim ersten Buchstaben des Texts an und Amira beim zweiten. Die Weite des nächsten Sprungs wird dabei vom „Sprungweitenwert“ des jeweiligen Buchstabens, auf dem sich der aktuelle Hopser befindet, bestimmt. Den Sprungweitenwert eines Buchstabens können wir in der folgenden Tabelle ablesen.

Buchstabe	a	b	c	...	x	y	z	ä	ö	ü	ß
Sprungweitenwert	1	2	3	...	24	25	26	27	28	29	30

Tabelle 1: Buchstaben und ihre zugehörigen Sprungweiten (Auszug)

Beim Hopsen ignorieren Bela und Amira sämtliche Symbole, die nicht in der Sprungweitentabelle (Tabelle 1) aufgeführt sind. Diese haben also keine Auswirkung auf die Weite eines Sprungs. Ausgenommen davon sind Großbuchstaben, deren zugehörigem Kleinbuchstaben eine Sprungweite zugeordnet ist. In dem Fall hopsen Amira oder Bela dann so weit wie der Sprungweitenwert des dazugehörigen Kleinbuchstabens. Anhand der Abbildung J2.1 können wir das Prinzip nochmal grafisch nachvollziehen: Alle ausgegrauten Zeichen, wie Zahlen, Leer- und Satzzeichen, wurden bei der Sprungweite nicht einbezogen.

Der␣43.␣Bundeswettbewerb␣Informatik

Abbildung J2.1: Beispielhaftes Überspringen von ignorierbaren Zeichen

Beim eigentlichen Texthopsen startet Amira auf dem zweiten Buchstaben des Texts, im Beispiel also auf dem *e*, und Bela auf dem ersten Buchstaben, hier das *D*. Sollte der gegebene Text mit einem oder mehreren Zeichen beginnen, denen keine Sprungweite zugeordnet ist, so werden diese ignoriert und die beiden Texthopsenden starten beim ersten beziehungsweise zweiten erlaubten Zeichen. Von dort aus hopsen sie dann durch den Text. Dafür nimmt derjenige, der aktuell am Zug ist, den Buchstaben an seiner aktuellen Position und hopst so viele Buchstaben weiter, wie der Sprungweitenwert dieses Buchstabens vorgibt. Der aktuelle Buchstabe selber wird dabei nicht in die Distanzberechnung miteinbezogen. Um also auf den nachfolgenden Buchstaben zu springen, muss die Sprungweite des aktuellen Buchstabens gleich 1 sein. Es handelt sich in diesem Fall beim aktuellen Buchstaben um ein *a*.

Die Prozedur des Hopsens wiederholen die beiden abwechselnd solange, bis beide aus dem Text heraus gehopst sind. „Aus einem Text heraushopsen“ bedeutet, dass die Sprungweite des aktuellen Buchstaben größer ist als die verbliebene Anzahl an erlaubten Zeichen im Text. Wäre der aktuelle Buchstabe beispielsweise ein *d* mit Sprungweite fünf, auf das *d* würden in dem Text aber nur noch drei weitere Buchstaben folgen, so würde mit dem Sprung ab dem *d* in diesem Beispiel aus dem Text heraus gehopst werden.

Die Anzahl an Hopsern, die Bela oder Amira in einem Text machen, wollen wir für den weiteren Verlauf als h_B für Belas Anzahl an Texthopsern und h_A für Amiras Anzahl bezeichnen.

Überlegungen zum Gewinn des Texthopsens

Da in der Aufgabenstellung nicht explizit erklärt ist, wann jemand „als Erstes“ aus dem Text heraushopst und somit gewinnt, liegt es an den Bearbeitenden, sich für eine Lesart des Gewinnkriteriums zu entscheiden. Grundsätzlich sind zwei verschiedene Interpretationen möglich:

1. **Wertung nach der Runde:** Amira und Bela bestimmen erst, nachdem beide gesprungen sind, ob jemand gewonnen hat. Um zu gewinnen, muss einer von beiden weniger Texthopser zum Durchqueren des Texts als der jeweils andere gebraucht haben ($h_A < h_B$ oder $h_B < h_A$). Hierbei ist ein Unentschieden möglich, wenn sowohl Amira als auch Bela in derselben Runde aus dem Text heraushopsen, also ihre Anzahl an Hopsern gleich ist ($h_A = h_B$).
2. **Wertung während der Runde:** Sobald entweder Amira oder Bela aus dem Text herausgehopst ist, endet das Spiel, auch wenn noch nicht beide in der entsprechenden Runde am Zug waren. Bei einer theoretisch gleichen Anzahl an benötigten Texthopsern würde dann die Person gewinnen, die zuerst am Zug ist; wenn Amira in jeder Runde zuerst hopst, gilt für Amiras Gewinn: $h_A \leq h_B$, wenn Bela als erstes hopst, gilt $h_B \leq h_A$ für seinen Gewinn. Aus diesem Grund ist es bei dieser Interpretation des Gewinnkriteriums wichtig zu definieren, ob Amira oder Bela jeweils zuerst hopst, da dies den Ausgang des Spiels beeinflusst.

J2.2 Umsetzung

Bei der Umsetzung des Texthopsens sollten ein paar Dinge beachtet werden, damit auch korrekt bestimmt werden kann, ob Amira oder Bela das Texthopsen gewinnt.

Zunächst muss der gegebene Text beim Einlesen so kodiert werden, dass *ä*'s, *ö*'s, *ü*'s und *ß*'s richtig repräsentiert werden. Eine hierfür geeignete Kodierung ist beispielsweise *UTF-8*[2]. Wird der Text beim Einlesen nicht richtig kodiert, ist es möglich, dass die Ermittlung der korrekten Sprungweiten deutlich erschwert wird bis hin zu unmöglich ist.

Weiterhin muss für jeden erlaubten Buchstaben die dazugehörige Sprungweite (vgl. Tabelle 1) ermittelbar sein. Dies kann entweder statisch durch eine geeignete Datenstruktur wie eine Map oder ein Array geschehen oder dynamisch. Bei der dynamischen Berechnung der Sprungweiten kann für die Buchstaben des lateinischen Alphabets (a, …, z) der jeweilige Index in der UTF8-Kodierung (a: 97, …, z: 122) genutzt und durch eine Subtraktion mit 96 entsprechend der Sprungweitentabelle normalisiert werden. Die dynamische Generierung der Sprungweiten für *ä*, *ö*, *ü* und *ß* (UTF-8-Indizes 228, 246, 252 und 223) gestaltet sich bei dieser Methodik eher schwierig, da die Indizes der Umlaute und *ß* nicht direkt auf die des lateinische Alphabets folgen und somit eine simple Normalisierung durch Subtraktion eines festen Werts nicht möglich ist.

Des Weiteren muss sich überlegt werden, wie mit nicht erlaubten Zeichen und Großbuchstaben verfahren werden soll: Da nur Kleinbuchstaben, aber nicht Großbuchstaben, eine Sprungweite zugeordnet ist, können entweder Großbuchstaben in die entsprechenden Kleinbuchstaben umgewandelt werden, oder unsere Repräsentation der Tabelle kann entsprechend für Großbuchstaben erweitert werden. Bei nicht erlaubten Zeichen ist es möglich, diese direkt im Text zu entfernen oder sie während des Hopsens auszulassen.

[2]Für weitere Informationen zu UTF-8 siehe bspw. `https://de.wikipedia.org/wiki/UTF-8`

Beim Umsetzen des Texthopsens selber sind verschiedene Varianten denkbar. Zum einen kann das Hopsen von Amira und Bela wie in der Aufgabenstellung beschrieben abwechselnd erfolgen. Je nach gewählter Interpretation des Gewinnkriteriums, bricht man das Hopsen ab, sobald jemand aus dem Text heraus gehopst ist. Zum anderen können wir das Hopsen nacheinander simulieren und dabei die Anzahl an Texthopsern für beide Spieler zählen. Dies geht, da Amiras Hopser keinen Einfluss auf Belas Hopser haben und umgekehrt. Am Ende können wir hierbei die Anzahl an Texthopsern von beiden je nach gewählter Interpretation des Gewinnkriteriums vergleichen.

J2.3 Beispiele

Je nach gewählter Interpretation des Gewinnkriteriums ist ein anderer Ausgang des Spiels möglich. Die Anzahl an Hopsern ist aber unabhängig vom Gewinnkriterium gleich.

hopsen1.txt

- Amira: 70 Texthopser
- Bela: 68 Texthopser

Bela gewinnt unabhängig von der Interpretation des Gewinnkriteriums.

hopsen2.txt

- Amira: 25 Texthopser
- Bela: 25 Texthopser

Je nach Interpretation gibt es ein Unentschieden, oder die Person, die begonnen hat, gewinnt.

hopsen3.txt

- Amira: 18 Texthopser
- Bela: 18 Texthopser

Je nach Interpretation gibt es ein Unentschieden, oder die Person, die begonnen hat, gewinnt.

hopsen4.txt

- Amira: 32 Texthopser
- Bela: 35 Texthopser

Amira gewinnt unabhängig von der Interpretation des Gewinnkriteriums.

hopsen5.txt

- Amira: 930 Texthopser
- Bela: 923 Texthopser

Bela gewinnt unabhängig von der Interpretation des Gewinnkriteriums.

J2.4 Bewertungskriterien

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

- [−1] **Gewinnkriterium nicht erkennbar**
  In der Aufgabenstellung ist definiert, dass gewinnt, wer als Erstes herausspringt. Es muss klar werden, ob dies als „die erste Person, die herausspringt“ oder „die Person mit weniger Sprüngen“ interpretiert wurde.
- [−1] **Modellierung ungeeignet**
  Beim Einlesen und Speichern des Textes muss ggf. auf die Kodierung der Datei geachtet werden; insbesondere müssen die Umlaute *ä*, *ö* und *ü* sowie das *ß* korrekt eingelesen werden. Es muss unkompliziert möglich sein, für jeden Buchstaben die Sprungweite zu ermitteln.
- [−1] **Lösungsverfahren fehlerhaft**
  Die Sprungfolgen müssen korrekt berechnet werden. Zeichen, die keine Buchstaben sind, dürfen bei den Sprüngen nicht mitgezählt werden. Insbesondere muss auf Großbuchstaben geachtet werden, aber für *ä*, *ö*, *ü* und *ß* ist je nach Sprungweiten-Ermittlung eine Zusatzbedingung notwendig.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Es muss mindestens ausgegeben werden, wer gewonnen hat oder ob es ein Unentschieden gab.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens vier der fünf vorgegebenen Beispiele `hopsen1.txt` bis `hopsen5.txt` enthalten.

	Anzahl Hopser		
Bsp.	**Amira**	**Bela**	**Ergebnis**
1	70	68	Bela gewinnt.
2	25	25	Unentschieden oder die beginnende Person gewinnt.
3	18	18	Unentschieden oder die beginnende Person gewinnt.
4	32	35	Amira gewinnt.
5	930	923	Bela gewinnt.

Aufgabe 1: Hopsitexte

1.1 Lösungsidee

Um Zara optimal beim Schreiben von Hopsitexten zu unterstützen, müssen wir uns vorher überlegen, was ein Programm zu diesem Zweck leisten sollte. Das Programm wird in Zaras Schreibprozess nur eine assistierende Rolle einnehmen, also keinen Inhalt selbst generieren, da Zara nur ein Programm benötigt, das ihr hilft, Hopsitexte zu schreiben.

Erkennung von Hopsitexten

Ein Programm, das bei der Erstellung von Hopsitexten unterstützen soll, muss zunächst einmal überhaupt erkennen können, ob es sich bei dem durch die Benutzenden eingegebenen Text um einen Hopsitext oder einen fehlerhaften Text, also keinen Hopsitext, handelt. Dafür ist eine Simulation des Texthopsens mit zwei Spielparteien durch den gegebenen Text sinnvoll. Bei der Simulation werden die verschiedenen Sprungpositionen der Spieler im Text ausgehend vom ersten und zweiten erlaubten Buchstaben im Text erfasst. Anhand dieser Daten ist es möglich, zu erkennen, ob der gegebene Text ein Hopsitext oder fehlerhaft ist. Springen nämlich die beiden Spieler von zwei unterschiedlichen Positionen aus auf dieselbe Position, gibt es also eine *Kollision*, sind ihre Sprungfolgen auch im weiteren Verlauf des Texts identisch. Dadurch würden sie auch ausgehend von derselben Stelle aus dem Text springen, was den eingegebenen Text zu einem fehlerhaften Text, also keinen Hopsitext machen würde. Ein Hopsitext ist somit daran erkennbar, dass sich die Sprungfolgen der beiden Spielparteien zu keinem Zeitpunkt überschneiden, es demnach keine Kollisionen im Text gibt.

Bedienbarkeit

Weiterhin sollte das Programm das Editieren von fehlerhaften Texten hin zu Hopsitexten unterstützen, indem es gut bedienbar ist. Eine gute Bedienbarkeit muss sich in unserem Programm weniger durch eine ausgeprägte grafische Bedienoberfläche äußern als durch ein wohl überlegtes Konzept zur Bedienung. Hierbei soll eher die Funktionsweise des Programms im Vordergrund stehen.

Zunächst muss unser Programm die Eingabe von Text durch die Benutzerin ermöglichen oder ihren Text einlesen können. Wenn dies nicht gegeben ist, kann das Programm Zara beim *Schreiben* von Hopsitexten gar nicht erst sinnvoll helfen. Des Weiteren muss ein Programm zur Unterstützung der Erstellung von Hopsitexten mindestens erkennen können, ob ein Text fehlerhaft oder ein Hopsitext ist. Diese Erkenntnis muss auch an die Benutzerin zurück gemeldet werden. Dennoch ist das Programm nur eingeschränkt hilfreich, wenn es Zara nur meldet, ob ihr eingegebener Text ein Hopsitext ist oder nicht. Damit Zara besonders gut mit dem Programm Hopsitexte erstellen kann, muss sie auch wissen, was die Ursache für die Kollision ist, also wo genau das Problem in ihrem Text ist. Dafür kann das Programm auf die bereits bei der Kollisionserkennung erstellten Sprungfolgen zurückgreifen. Die Positionen der beiden Spielparteien direkt vor der Kollision sind nämlich ursächlich für diese. Wird mindestens eines der Zeichen an den problematischen Stellen entsprechend angepasst, wird die von diesen Stellen ausgelöste Kollision aufgelöst. Natürlich ist dadurch nicht garantiert, dass alle Kollisionen in dem Text aufgelöst sind oder dass an anderer Stelle eine neue Kollision entsteht, aber zumindest die offensichtlich bestehende Kollision in dem gegebenen Text ist aufgelöst.

Neben den genannten Grundfunktionen, die die Mindestanforderungen an unser Programm darstellen, sind noch einige Erweiterungen denkbar. Jedoch sind solche Erweiterungen nicht von den Teilnehmenden der 1. Runde gefordert und würden den Rahmen dieses Dokuments sprengen.

1.2 Umsetzung

Die Umsetzung einer Simulation des Texthopsens zur Erkennung von Kollisionen kann parallel zur Umsetzung des Texthopsens in Junioraufgabe 2 erfolgen. Jedoch sollten die Sprungfolgen in einer geeigneten Datenstruktur gespeichert werden, damit diese nachverfolgt und die Ursachen für Kollisionen erkannt werden können. Wenn mit einem interpunktierten Text beispielsweise in der Ausgabe gearbeitet werden soll, ist es entweder nötig, das Texthopsen im Text mit Interpunktionen zu simulieren oder eine Version des Texts ohne Interpunktion zum Hopsen und eine mit Interpunktion vorzuhalten.

Für die Umsetzung der Kollisionserkennung und der Kollisionsursachennachverfolgung sind grob zwei Methodiken zu unterscheiden: Zum einen können die kompletten Sprungfolgen der beiden Spielparteien, beispielsweise durch farbliche Hervorhebung, nachvollziehbar gemacht werden. Zum anderen kann nur die Kollision und ihre Ursache nachvollziehbar gemacht werden. Dies kann beispielsweise durch Angabe der ursächlichen Zeichen und ihrer Position im Text umgesetzt werden.

Dies ist ein Beispiel für einen Hopsitext ohne Kollision.

Abbildung 1.1: Farblich hervorgehobene Sprungfolgen in einem Text ohne Kollision. Im Falle einer Kollision würde diese in rot hervorgehoben werden.

1.3 Beispiele

Die Dateien `hopsen1.txt` und `hopsen2.txt` entsprechen den gleichnamigen Beispieleingaben aus der Junioraufgabe 2 (Texthopsen).

Beispiel 1

Bei dieser beispielhaften Umsetzung wird zunächst der Pfad zu einer Textdatei eingegeben. Daraufhin überprüft das Programm, ob es sich bei dem gegebenen Text aus der Datei um einen Hopsitext handelt. Ist das der Fall, wird eine entsprechende Meldung ausgegeben (Abbildung 1.2). Wenn der Text fehlerhaft ist (Abbildung 1.3), werden sowohl die Indizes (ausgehend von 1) der betroffenen Buchstaben ausgegeben als auch ein farbig markierter Ausschnitt. Bei der farblichen Markierung sind die Positionen der Spielparteien direkt vor der Kollision rot und blau, die Kollision selber ist in pink eingefärbt. Die Klammern unter dem Textausschnitt sollen die Sprünge, die zur Kollision führen, nochmal auf eine andere Art visualisieren.

```
Datei
> hopsen1.txt
Text ist ein Hopsitext.
```

Abbildung 1.2: `hopsen1.txt`: Dieser Text enthält keine Kollision und ist daher ein Hopsitext.

```
Datei
> hopsen2.txt
Kein Hopsitext!

Kollision an "l" (52. Zeichen) ausgehend von "s" (28. Zeichen) bzw. "d" (47. Zeichen)
 ... das Land; Ein Nilpferd schlum ...
```

Abbildung 1.3: `hopsen2.txt`: Bei diesem Text kollidieren die Sprungfolgen an einem *l*, dem 52. Zeichen im Text, ausgehend von einem *s* (28. Zeichen) und einem *d* (47. Zeichen). Um Platz bei der Ausgabe zu sparen, wird nur der betroffene Ausschnitt des Texts ausgegeben.

Beispiel 2

Im zweiten Beispiel ist es möglich, in einem Feld den Text direkt einzutippen und unten die farblichen Markierungen der Sprünge zu sehen. Die Farben der Spielparteien sind dabei grün und blau, während kollidierende Positionen rot eingefärbt sind.

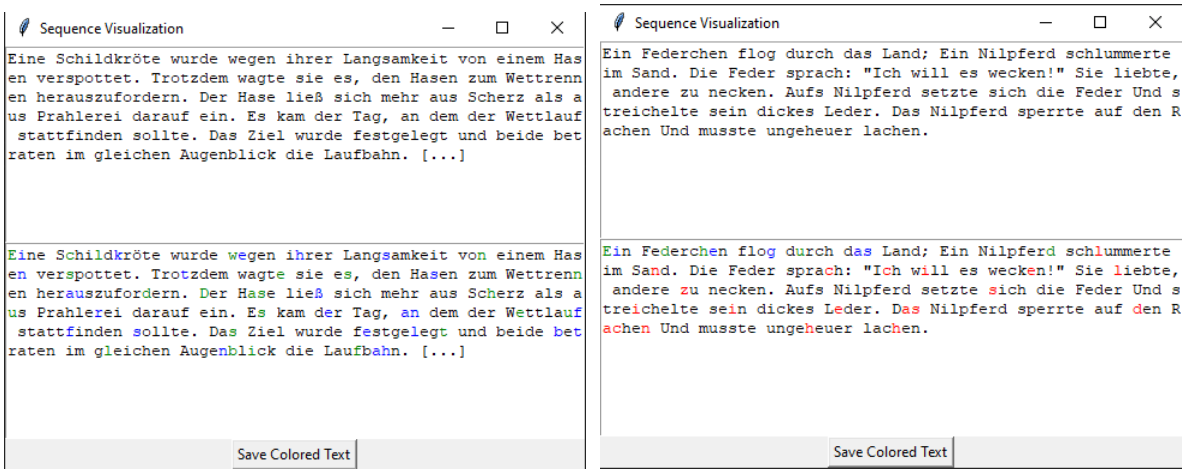


Abbildung 1.4: Links `hopsen1.txt` und rechts `hopsen2.txt`.

1.4 Bewertungskriterien

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

- $[-1]$ **Texthopsen nicht korrekt simuliert**
  Beim Einlesen und Speichern des Textes muss darauf geachtet werden, dass die Umlaute *ä*, *ö*, *ü* und das *ß* korrekt behandelt werden. Für Buchstaben muss deren Sprungweite korrekt ermittelt werden. Die Sprungfolgen müssen korrekt berechnet werden.
- $[-1]$ **Assistenzfähigkeit des Programmes unzureichend begründet**
  Es sind Überlegungen gefordert, was das Programm bieten muss, um jemand beim Erstellen von Hopsitexten zu assistieren. Es muss aus der Dokumentation ersichtlich werden, wie man mit dem Programm arbeiten kann, welche Assistenzfunktionen es hat und warum diese hilfreich sind.
- $[-1]$ **Assistenzprogramm erfüllt nicht die Anforderungen**
  Das Assistenzprogramm muss die beschriebenen Assistenzfunktionen korrekt umsetzen. Insbesondere muss es mindestens
  - einen Text verarbeiten und
  - eine Kollision korrekt erkennen und zurückmelden können.
- $[-1]$ **Texte nicht sinnvoll**
  Es ist gefordert, dass „sinnvolle deutsche“ Texte entstehen. Wenn das Assistenzprogramm dies unmöglich macht, indem bspw. bei einer Kollision Buchstaben gelöscht oder zufällige Worte aneinander gereiht werden, so wird ein Punkt abgezogen.
- $[-1]$ **Assistenzfähigkeit des Programmes schlecht nachvollziehbar**
  Es muss aus den Ausgaben des Programms klar ersichtlich werden, dass und warum ein Text kein Hopsitext ist. Dies kann bspw. durch klares Aufzeigen von Kollisionen sein. Dies gilt nicht für Programme, die von vornherein nur Hopsitexte generieren.
- $[-1]$ **Assistenzfähigkeit des Programmes unzureichend dokumentiert**
  Die Assistenzfähigkeit muss anhand mindestens eines Beispieltextes erläutert werden. Die Benutzung des Programms muss mindestens je einmal anhand eines korrekten Hopsitexts und eines fehlerhaften Texts dokumentiert sein.

Aufgabe 2: Schwierigkeiten

2.1 Lösungsidee

In dieser Aufgabe geht es darum, die Schwierigkeitsabstufungen (im Folgenden *Anordnungen*) der m Klausuraufgaben, die durch die Altklausuren vorgegeben sind, zu nutzen, um eine mögliche Anordnung einer gegebenen Menge an k Klausuraufgaben zu berechnen, die konsistent mit den Anordnungen in den Altklausuren ist. Dabei können Konflikte in den Anordnungen der Altklausuren auftreten, die eine besondere Behandlung erfordern.

Beobachtungen

Zunächst fällt auf, dass es in einigen Fällen mehrere mögliche Lösungen geben kann. Wenn beispielsweise `A < B` und `A < C` gegeben sind, kann eine valide Anordnung für `A`, `B` und `C` entweder `A, B, C` oder `A, C, B` lauten. Die Lösung ist also nicht immer eindeutig. Außerdem lassen sich einige Beobachtungen in Bezug auf *Konflikte* machen. Konflikte können auf viele Weisen entstehen. Es könnte beispielsweise eine zyklische *leichter-als* Beziehung zwischen 3 Aufgaben entstehen (also z.B. `A < B`, `B < C`, `C < A`). Es fällt also auf, dass Konflikte zwischen beliebig vielen Aufgaben entstehen können und ein Konflikt insofern als eine Eigenschaft zu verstehen ist, welche eine Gruppe von Aufgaben betrifft; eine Aufgabe kann also *Teil* eines größeren Konflikts sein. Außerdem bestehen bei genauerer Betrachtung noch einige kompliziertere Möglichkeiten für Konflikte (wie zum Beispiel überschneidende Zyklen).

Das Erkennen dieser Konflikte stellt eine Hauptschwierigkeit dieser Aufgabe dar. Wir können zudem eine leichte Änderung des Problems vornehmen. Wir werden jeweils das Problem des Findens einer Anordnung direkt für alle Aufgaben lösen, statt nur für die gegebene Teilmenge. Es kann im schlimmsten Fall immer sein, dass die gegebene Teilmenge der Gesamtmenge an Aufgaben entspricht. Somit ist das Problem der Anordnung einer Teilmenge im Allgemeinen mindestens so schwierig wie das Finden einer Anordnung aller Aufgaben. Gleichzeitig ist es auch maximal so schwierig, da wir aus einer Anordnung aller Aufgaben einfach die Anordnung der Teilmenge extrahieren können, indem wir alle nicht relevanten Aufgaben in der Anordnung ignorieren. Insofern werden wir also im Folgenden direkt eine Lösung für alle m Aufgaben finden.

Brute Force

Ein erster naiver Ansatz könnte darin bestehen, alle möglichen Anordnungen (Permutationen) der gegebenen Menge an k Aufgaben auszuprobieren und jeweils zu überprüfen, ob diese zu einer gegebenen Anordnung im Widerspruch steht (hierbei könnte es noch nützlich sein, nur k Aufgaben zu ordnen; wie bereits beschrieben, gilt im schlimmsten Fall jedoch $k = m$). Unabhängig davon, wie man dabei vorgeht, um Konflikte festzustellen, und wie effizient man die Überprüfung der Konsistenz mit den gegebenen Klausuren gestaltet, bekommt man jedoch Probleme aus Laufzeitperspektive, da es für k Aufgaben insgesamt $k!$ mögliche Permutationen gibt. Speziell für die letzte Beispieleingabe ist dieser Ansatz also nicht ausreichend. Das grundlegende Problem im beschriebenen Ansatz ist, dass keine der gegebenen Informationen (also Anordnungen aus Altklausuren) wirklich verwendet wird, um die Anordnungen zu erstellen, sondern Anordnungen naiv generiert und nur folgend mit den gegebenen Informationen auf

Konsistenz überprüft werden. Die Schwierigkeit und Idee ist also, die gegebenen Informationen auf geeignete Weise zu nutzen, um daraus eine konsistente Anordnung zu berechnen. Von hier lassen sich verschiedene Ansätze ausführen.

Im Folgenden soll zunächst die Modellierung als Graph[3], der Umgang mit Konflikten im Rahmen dieser Modellierung, ein Lösungsansatz ohne explizite graphentheoretische Algorithmen und ein etwas effizienterer Ansatz mithilfe von Graphenalgorithmen diskutiert werden. Andere Ansätze welche keine explizite Graphenmodellierung verwenden, laufen womöglich ebenfalls auf graphenähnliche Strukturen und Ideen hinaus. Insofern ist es sehr lehrreich, sich diese Ansätze anzuschauen. Graphentheorie lässt sich vielseitig anwenden. Unter anderem in Problemen, bei welchen es bestimmte Relationen (also Beziehungen) zwischen bestimmten Objekten (hier Aufgaben) gibt, kann eine Modellierung als Graph und eine *Übersetzung* des Problems in ein graphentheoretisches Problem sehr hilfreich sein. Es gibt viele bekannte Strukturen und Algorithmen, welche sich womöglich auf das gegebene Problem anwenden lassen.

Modellierung als Graph & Umgang mit Konflikten

Um Dua L. Graph zu helfen, lässt der Sachzusammenhang sich als gerichteter Graph modellieren. Jede gegebene Aufgabe entspricht dabei einem Knoten und für jede gegebene *leichter als*-Relation zweier Aufgaben eine gerichtete Kante. Für das Beispiel aus der Aufgabenstellung erhält man also den folgenden Graph:

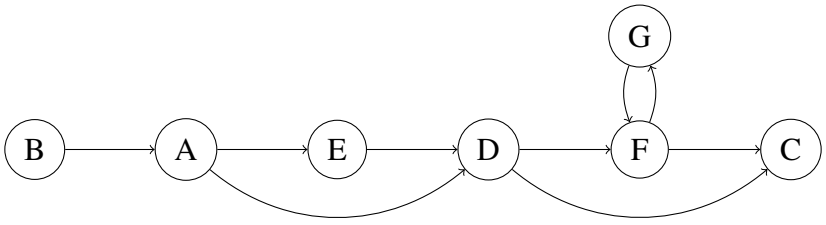


Abbildung 2.1: Modellierung des Sachzusammenhangs aus der Aufgabenstellung als Graph

Wir werden im Folgenden häufig die Begriffe Knoten und Aufgaben gleichbedeutend verwenden. Man beachte, dass lediglich Kanten zwischen in der Eingabe direkt aufeinanderfolgenden Aufgaben eingefügt werden. Die Transitivität (also, dass `A < B < C` bedeutet, dass ebenfalls `A < C`) wird also nicht explizit im Graph modelliert. Wir bezeichnen *leichter-als* Relationen zweier Aufgaben, welche in der Eingabe direkt aufeinanderfolgen, als *direkt leichtere* bzw. *direkt schwierigere* Aufgabe. In der Graphenmodellierung werden also nur Kanten für solche *direkte Relationen* eingefügt.

Nun wollen wir überlegen wie man mit Konflikten zweier Aufgaben umgehen kann. In diesem Lösungsansatz soll die folgende natürliche Interpretation von Konflikten diskutiert werden: Zwei Aufgaben X und Y, welche im Konflikt stehen, werden wir im Kontext als *gleichschwierig* betrachten, somit werden wir die Anordnungen $\langle \ldots, X, Y, \ldots \rangle$ und $\langle \ldots, Y, X, \ldots \rangle$ als gerechtfertigt angesehen. Wir bezeichnen zwei *gleich schwierige* Aufgaben im Folgenden mit $X \sim Y$. Man kann sich davon überzeugen, dass dies in unserer Modellierung genau dem Fall entspricht, dass ein Pfad von X nach Y und ein Pfad von Y nach X existiert. Man beachte, dass ein Pfad

[3] `https://soi.ch/wiki/graphs/`

mehrere Knoten beinhalten kann. Interessant ist hierbei, dass falls $X \sim Y$ und $Y \sim Z$, auch $X \sim Z$ gelten muss (dies kann man sich anhand der Pfade in der Graphenmodellierung überlegen): Falls ein Pfad von X nach Y existiert und ein Pfad von von Y nach Z existiert, existiert auch ein Pfad von X nach Z und andersherum. Formal handelt es sich bei $\sim$ um eine sogenannte Äquivalenzrelation[4].

Intuitiv betrachten wir also *Gruppierungen* (im folgenden Äquivalenzklassen) von *gleichschwierigen* Aufgaben, wobei zwischen zwei beliebigen Aufgaben einer *Gruppierung* Pfade in beide Richtungen existieren. Damit wir eine konsistente Modellierung haben, betrachten wir auch Aufgaben, welche nicht in Konflikt stehen, als Teil einer jeweils eigenen Äquivalenzklasse. Wir müssen also die folgenden beiden Teilprobleme lösen:

1. Alle beschriebenen Äquivalenzklassen bestimmen (bzw. *Gruppierungen* mit Aufgaben *gleicher Schwierigkeit*)
2. Die Äquivalenzklassen ihrer Schwierigkeit nach anordnen. Innerhalb der Äquivalenzklassen ist uns die Anordnung der Aufgaben also egal. Lediglich die Anordnung der Äquivalenzklassen ist zu berechnen.

In der Graphentheorie bezeichnet man dieses Konzept der beidseitigen Erreichbarkeit in gerichteten Graphen durch Pfade auch als starke Zusammenhangskomponenten[5], welche hierbei unseren Äquivalenzklassen entsprechen. Es existieren effiziente Algorithmen, mit denen sich diese Komponenten berechnen lassen (siehe Ansatz mit expliziter Graphentheorie).

Erster Ansatz: Mengen von leichteren Aufgaben

Ein Ansatz könnte darin bestehen, dass man aus der Eingabe für jede Aufgabe X eine Menge S_X von allen *leichteren* Aufgaben konstruiert. Es gibt verschiedene Wege, wie man die Eingabe einlesen kann, um die Mengen S_X zu berechnen. Beispielsweise könnte man so vorgehen wie in Algorithmus 2. Wir gehen dabei davon aus, dass die *direkten Relationen* in einer bestimmten listenähnlichen Datenstruktur nach dem Einlesen vorliegen. Hierbei ist wichtig, dass die Mengen auch in der Implementierung Elemente wirklich nur einmal beinhalten können, da man ansonsten in eine Endlosschleife geraten könnte.

Algorithmus 2 Berechnung der Mengen S_X

```
Initialisiere für jede Aufgabe X eine leere Menge S_X
Für jede direkte Relation (X < Y):
    füge X zu S_Y hinzu
Wiederhole so lange, bis sich keine Mengen mehr ändern
    Für jede direkte Relation (X < Y):
        Für jede Aufgabe Z in S_X:
            Füge Z zu S_Y hinzu
```

Wenn wir jedoch unsere Graphenmodellierung nutzen, also die Eingabe einlesen und beispielsweise mit Adjazenzlisten als Graph modellieren[6], dann ist ein Aufgabe *leichter als* alle von dem zugehörigen Knoten durch Pfade erreichbaren Aufgaben. Algorithmisch kann die Berechnung

[4] `https://de.wikipedia.org/wiki/ÃĎquivalenzrelation`
[5] `https://soi.ch/wiki/scc/`
[6] `https://de.wikipedia.org/wiki/Adjazenzliste`

von S_X also realisiert werden, indem wir von jedem Knoten einmal *loslaufen* und den betrachteten Knoten bei allen erreichbaren Aufgaben in die Menge S_X einfügen (hierbei lassen sich erste Ideen der Graphentheorie erkennen[7]).

Algorithmus 3 Ansatz zur Berechnung der Äquivalenzklassen

```
c := mit -1 initialisiertes Array der Länge m ▷ Einträge representieren die Äq. der Aufgaben
t ← 0                                ▷ Zähler für die Anzahl bereits berechneter Äquivalenzklassen
for i = 1,...,m do                                          ▷ Iteration über alle Aufgaben
    if c[i] = −1 then          ▷ Die betrachtete Aufgabe ist noch keiner Äq. zugeordnet
        t ← t + 1
        c[i] ← t                                    ▷ Zuordnen einer neuen Äquivalenzklasse
        for X_j ∈ S_{X_i} do
            if X_i ∈ S_{X_j} then                          ▷ Überprüfung ob X_j ∼ X_i
                c[j] ← c[i]
            end if
        end for
    end if
end for
```

Die Äquivalenzklassen können nun auf Basis der Mengen S_X beispielsweise mit Algorithmus 3berechnet werden. Man beachte, dass man mit diesem Algorithmus, sobald man eine Aufgabe einer Äquivalenzklasse durchlaufen hat, für alle Knoten dieser Äquivalenzklasse die zugehörige Identifikation in das Array c geschrieben hat. Somit wird man die innere Schleife höchtens einmal für jede Äquivalenzklasse durchlaufen. Man nutzt hierbei aus, dass zwei Aufgaben X_i und X_j genau dann in der gleichen Äquivalenzklasse sind, wenn $X_i \in S_{X_j}$ und $X_j \in S_{X_i}$.

Der zweite Schritt besteht nun darin, die Äquivalenzklassen der Aufgaben ihrer Schwierigkeit anzuordnen. Zunächst einmal ist klar, dass es zwischen zwei Aufgaben, welche sich in verschiedenen Äquivalenzklassen befinden, keine Konflikte geben kann. Wir erhalten nun für jede *direkte Relation* `X < Y`, für welche sich X und Y in verschiedenen Äquivalenzklassen befinden, dass die Äquivalenzklasse von X *direkt leichter ist* als die Äquivalenzklasse von Y. Man kann sich nun davon überzeugen, dass es eine leichteste Äquivalenzklasse geben muss, da es einen Konflikt geben müsste, wenn eine solche nicht existiert.

Algorithmus 4 Berechnung einer Anordnung der Äquivalenzklassen

Sei $l[\alpha]$ für jede Äquivalenzklasse α, die Anzahl an *direkt leichteren* Äquivalenzklassen
Man initialisiere eine leere Warteschlange Q
Füge alle Äquivalenzklassen mit $l[\alpha] = 0$ in Q hinzu
Solange Q nicht leer ist:
    entferne die vorderste Äquivalenzklasse α aus Q
    Füge die Aufgaben aus α als nächst-schwierigere Aufgaben unserer Anordnung hinzu
    Für jede von α *direkt schwierigere* Äquivalenzklasse β:
        $l[\beta] \leftarrow l[\beta] - 1$
        Falls $l[\beta] = 0$, füge β der Warteschlange Q hinzu

[7] beispielsweise `https://de.wikipedia.org/wiki/Tiefensuche`

Algorithmus 4 kann nun genutzt werden um eine Anordnung der Äquivalenzklassen zu berechnen. Warum dieser Algorithmus die Klassen korrekt nach Schwierigkeit anordnet, wird durch folgende Überlegung klar: Zu Beginn starten wir sicher mit den Klassen, die als leichteste gelten, da diese keine *direkt leichteren Klassen* haben. Durch das Aktualisieren von $l[\beta]$ für die schwierigeren Klassen entfernt der Algorithmus effektiv die bereits angeordneten Klassen aus der weiteren Betrachtung. Dieser Schritt stellt sicher, dass nach jeder Iteration nur diejenigen Klassen als nächstes betrachtet werden, deren leichtere Vorgänger bereits verarbeitet wurden. Auf diese Weise wiederholt sich der Prozess, bis alle Klassen in der korrekten Reihenfolge nach Schwierigkeit angeordnet sind. Tatsächlich entspricht diese Idee einer sehr grundlegenden Idee der Graphentheorie: der **topologischen Sortierung**[8]. Derselbe Algorihmus wird in 5 nochmal etwas präziser und formaler auf Basis von graphentheoretischen Überlegungen formuliert; die Idee ist jedoch exakt die gleiche.

Effiziente Lösung mit Graphentheorie

Es existieren zwei sehr bekannte Algorithmen zur Berechnung starker Zusammenhangskomponenten in einem gerichteten Graphen: Kosarajus Algorithmus[9] und Tarjans Tiefensuche[10]. Die detaillierte Umsetzung und Funktionsweise dieser Algorithmen soll hierbei nicht weiter diskutiert werden. Es lohnt sich jedoch sehr, einmal im Internet nachzulesen. Im Vergleich zum ersten Ansatz lassen sich die starken Zusammenhangskomponenten bzw. Äquivalenzklassen mit diesen Algorithmen deutlich effizienter berechnen (siehe Laufzeitanalyse).

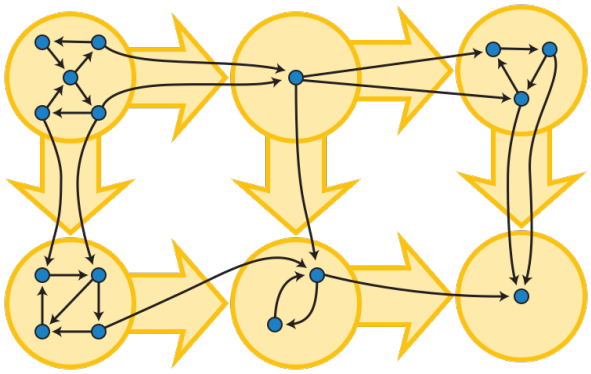


Abbildung 2.2: Kondensationsgraph - neuer gerichteter Graph wird auf den Zusammenhangskomponenten definiert.
Von David Eppstein - Eigenes Werk, CC0, `https://commons.wikimedia.org/w/index.php?curid=48917452`

Bis jetzt konnten wir also die starken Zusammenhangskomponenten in unserem Graphen berechnen, welche den Äquivalenzklassen entsprechen. Nun kommt eine zentrale Idee, welche sich durch Abbildung 2.2 visualisieren lässt (im Prinzip hatten wir diese Idee auch schon vorher, indem wir die *direkten Relationen* in Bezug auf die Äquivalenzklassen betrachtet haben). Nachdem wir die starken Zusammenhangskomponenten berechnet haben, lässt sich beobachten, dass wir im Grunde einen neuen Graphen erhalten. Wir wandeln dabei jede starke Zusammenhangskomponente in einen neuen Knoten um und erstellen jeweils eine gerichtete Kante von

[8] `https://soi.ch/wiki/toposort/`
[9] `https://soi.ch/wiki/scc/#algorithm-by-kosaraju`
[10] `https://de.wikipedia.org/wiki/Algorithmus_von_Tarjan_zur_Bestimmung_starker_Zusammenhangskomponenten`

Komponente K_1 zu K_2, falls Aufgaben $u \in K_1$ und $v \in K_2$ existieren, sodass eine Kante von u nach v im ursprünglichen Graph besteht. Der so erhaltene Graph (im folgenden Kondensationsgraph) hat eine wichtige Eigenschaft, welche intuitiv ebenfalls aus unserer Definition der Äquivalenzklassen auch bereits folgte: Er enthält keine Zyklen[11], bzw. im Sachzusammenhang ist es der Fall, dass keine zwei Aufgaben u und v existieren, welche wir verschiedenen Äquivalenzklassen zugeordnet haben, sodass ein Pfad von u nach v und von v nach u existiert. Wäre dies der Fall, dann würden u und v derselben Äquivalenzklasse angehören (das entspricht der vorherigen Beobachtung, dass eine *leichteste* Äquivalenzklasse existiert).

Das übrige Problem lautet also: Man finde eine Sortierung der Komponenten $\langle K_1, K_2, \ldots, K_l \rangle$ (sei l die Anzahl an Komponenten), sodass für alle $1 \leq i < j \leq n$ gilt, dass kein gerichteter Pfad im Kondensationsgraph von K_j nach K_i existiert. Ein gerichteter Pfad von K_j nach K_i entspricht mit unserer Modellierung jeweils der Situation im Sachzusammenhang, dass eine Aufgabe in der Äquivalenzklasse K_j *leichter als* eine Klausur in der Äquivalenzklasse K_i ist. Man beachte, dass wir durch die Beschreibung mithilfe von Pfaden (welche möglicherweise mehrere Knoten beinhaltet) die Transitivität der Relation betrachten. Hierfür existieren einige bekannte Algorithmen (wie bereits geschildert, handelt es sich hierbei um das Problem der topologischen Sortierung).

Algorithmus 5 Finden einer topologischen Sortierung

$A \leftarrow \langle\rangle$ ▷ leere Anordnung
$c[i] \leftarrow$ *Anzahl an eingehenden Kanten an Knoten* K_i
$Q \leftarrow$ leere Warteschlange
Füge in Q alle Knoten hinzu, welche keine eingehenden Kanten haben
while Q nicht leer **do**
    $K_i \leftarrow$ erster Knoten aus Q
    füge K_i der Anordnung A an
    for Nachbar[12] K_j von K_i **do**
        $c[j] \leftarrow c[j] - 1$
        if $c[j] = 0$ **then**
            Füge K_j in die Warteschlange Q ein
        end if
    end for
end while
A enthält die topologische Sortierung

Algorithmus 5 (auch als *Kahn's Algorithmus* bekannt) ist eine Umsetzung des zunächst beschriebenen Ansatzes im Kontext der Äquivalenzklassen. Im Grunde tut dieser genau das gleiche, bloß dass wir nun die expliziten graphentheoretischen Bezeichnungen nutzen. Mit diesem Algorithmus erhalten wir also direkt eine topologische Sortierung der Komponenten. Um die finale legale Anordnung gemäß unserer Handhabung von Konflikten zu generieren, müssen wir lediglich die irrelevanten, also nicht in der zu betrachtenden Teilmenge enthaltenen Aufgaben entfernen und können die Aufgaben innerhalb einer Komponente beliebig anordnen.

[11] `https://de.wikipedia.org/wiki/Zyklus_(Graphentheorie)`
[12] Hiermit werden Knoten bezeichnet, welche durch eine ausgehende Kante von K_i erreichbar sind

Laufzeit

Seien wie in der Eingabebeschreibung:

1. m die Gesamtanzahl an Aufgaben,
2. n die Anzahl an Klausuren und
3. k die Größe der Teilmenge der anzuordnenden Aufgaben.

Die Parameter k und n sind bei unserer Betrachtung eher irrelevant, da wir weiterhin davon ausgehen werden, dass wir eine Sortierung aller Aufgaben finden wollen. Außerdem spielt die Anzahl an Klausuren für unsere gewählten Ansätze auch nur beim Lesen der Eingabe eine Rolle. Im Extremfall könnte eine Anordnung der gesamten m Aufgaben in jeder der n Klausuren vorkommen. Somit würden wir im Extremfall eine Laufzeit für das Einlesen der Eingabe von $O(n \cdot m)$ haben.

Der interessante Teil der Laufzeitanalyse ist jedoch die algorithmische Umsetzung. Zunächst wollen wir den ersten Ansatz über die Mengen an leichteren Aufgaben und die Äquivalenzklassen an Aufgaben betrachten.

1. Algorithmus zum Berechnen der Mengen S_X. Hierbei ist interessant, wie oft wir im schlimmsten Fall die Schleife durchlaufen müssen, bis sich nichts mehr ändert. Man stelle sich hierfür vor, wie viele Iterationen es maximal dauern kann, damit eine Aufgabe über die Mengen S_X bis hin zu einer anderen Aufgabe *propagiert*. In jeder Iteration wird die Aufgabe der nächsten Menge S_X hinzugefügt. Da es insgesamt m Aufgaben gibt, kann es maximal m *Stationen* für eine solche Aufgabe geben. Somit wird die äußerste Schleife maximal m Iterationen brauchen. Innerhalb dieser Schleife iterieren wir jedoch über alle direkten Relationen (davon kann es im schlimmsten Fall $O(m^2)$ viele geben). Die Anzahl an Aufgaben in S_X ist durch m beschränkt, für die innerste Schleife erhalten wir dadurch eine weitere Schranke von m. Insgesamt erhalten wir also eine Laufzeit von $O(m^4)$.
2. Für die Berechnung von Äquivalenzklassen erhalten wir eine Laufzeit $O(m^2)$, da wir die beiden verschachtelten Schleifen jeweils durch m abschätzen können (mit cleverer Überlegung könnte man hier noch etwas besser abschätzen, was jedoch aufgrund der schlechten Laufzeit im ersten Schritt redundant ist).
3. Im letzten Schritt erhalten wir ebenfalls eine Laufzeit von $O(m^2)$, da wir im schlimmsten Fall m Äquivalenzklassen haben, welche alle *direkt leichter* als jede andere Äquivalenzklasse sind.

Insgesamt erhalten wir hier also eine Laufzeit von $O(m^4)$ unter Vernachlässigung vom Einlesen der Eingabe (dominiert durch den ersten Schritt). Mit dem Einlesen der Eingabe erhalten wir: $O(n \cdot m + m^4)$.

Wenn wir den obigen Graphenansatz mit *Tarjan's Tiefensuche* für das Finden der *starken Zusammenhangskomponenten* und *Kahn's Algorithmus* für die topologische Sortierung der *starken Zusammenhangskomponenten* nutzen, dann erhalten wir für diesen Teil eine Laufzeit von $O(\#\text{Knoten} + \#\text{Kanten})$ (Laufzeit der Algorithmen). Da wir insgesamt höchstens m Knoten und m^2 Kanten in unserem Graphen haben werden, erhalten wir eine Gesamtlaufzeit (mit Einlesen der Eingabe) der Ordnung $O(n \cdot m + m^2)$. Interessant ist jedoch, dass wir es tatsächlich schaffen, den algorithmischen Teil des Problems in $O(\#\text{Knoten} + \#\text{Kanten})$ (entsprechend unserer Graphenmodellierung) zu lösen, was im Kontext von Graphentheorie häufig einer unteren Schranke entspricht, da wir mindestens diese Zeit benötigen, um einen Graphen zu traversieren.

2.2 Beispiele

Wenn man Aufgaben, welche im Konflikt stehen als *gleich schwierig* betrachtet, dann gibt es in vielen Fällen mehrere korrekte Lösungen. Es wird die folgende Syntax für Ambiguität in den möglichen Lösungen verwendet:

1. `[A|B]`, bedeutet, dass an dieser Stelle Aufgabe `A` *oder* `B` möglich ist.
2. $[\overline{\texttt{AB}}]$, bedeutet, dass an dieser Stelle eine beliebige Permutation von `AB` möglich ist (dies ist also äquivalent zu `[AB|BA]`)

Mit Verschachtelungen (Interleavings) von Teilanordnungen werden im Folgenden beliebige Anordnungen bezeichnet, welche die Teilanordnungen respektieren. Die möglichen Interleavings der beiden Teilanordnungen

1. `A B`
2. `C D`

sind also beispielsweise: `A B C D` / `A C B D` / `A C D B` / `C A B D` / `C A D B` / `C D A B`
Wenn l_i die Länge der i-ten Teilanordnung ist, gibt es für das Interleaving also $(\sum_i l_i)!/\prod_i l_i!$ Möglichkeiten.

schwierigkeiten0.txt

Dieses Beispiel entspricht dem Sachverhalt aus der Aufgabenstellung. Es existiert hierbei nur eine mögliche Ausgabe:
`B E D F C`

schwierigkeiten1.txt

In diesem Beispiel gibt es keine *Konflikte*. Es gibt dennoch zwei mögliche Ausgaben:
`A` $[\overline{\texttt{CG}}]$ `F D`

schwierigkeiten2.txt

In diesem Beispiel kommt es darauf an wie man die *Konflikte* handhabt. Wenn man zwei Aufgaben welche im Konflikt stehen als gleichschwierig einstuft und die Anordnung gleichschwieriger Aufgaben somit als „egal“ betrachtet, dann sind alle Ausgaben der folgenden Form legal:
$[\overline{\texttt{ABDE}}][\overline{\texttt{FG}}]$
Also beispielsweise:
`A B D E F G`

schwierigkeiten3.txt

Unter derselben Annahme wie im vorherigen Beispiel (Umgang mit *Konflikten*), ist jede beliebige Verschachtelung (Interleaving) der folgenden 3 Teilanordnungen legal:

1. $[\overline{\texttt{ABCD}}]$ `E F G`
2. $\overline{\texttt{MN}}$
3. $[\overline{\texttt{HIL}}]$ `J K`

Also beispielsweise: `A B C D E F G M N H I L J K`

schwierigkeiten4.txt

Die einzig mögliche Ausgabe ist
`B I F N W`

schwierigkeiten5.txt

Unter derselben Annahme wie in Beispiel 2 und 3 (Umgang mit *Konflikten*), betrachte man eine beliebige Verschachtelung $\mathscr{A}$ (Interleaving) der folgenden 3 Teilanordnungen:

1. `H [SC] E N [OM] J L F V`
2. `R`
3. `Z Q K`

Alle korrekten Ausgaben haben nun die Form:
$\mathscr{A}$ `[GPDTUXA] B W I Y`
Also beispielsweise: `H R Z S C Q E K N M O J L F V G P D T U X A B W I Y`

2.3 Bewertungskriterien

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

- [−1] **Konflikte unzureichend behandelt**
  Wir erwarten keine formale Benennung der Konflikte, aber die Handhabung von Konflikten muss erklärt werden und sinnvoll sein.
- [−1] **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
- [−1] **Modellierung ungeeignet**
  Es darf nicht unnötig kompliziert sein, die Relationen der Klausuren zu überprüfen. Insbesondere muss auch die Transitivität innerhalb von Altklausuren erkannt und berücksichtigt werden. Wird die Transitivität zwischen Aufgaben aus verschiedenen Altklausuren nicht berücksichtigt, werden keine Punkte abgezogen.
- [−1] **Lösungsverfahren fehlerhaft**
  Das Programm muss eine *gute* Anordnung gemäß der gegebenen Altklausuren ausgeben. Insbesondere müssen alle Anordnungen von Aufgaben, welche nicht im Konflikt stehen, konsistent mit den Anordnungen aus den Altklausuren sein. Es müssen die geforderten k Aufgaben in der finalen Anordnung enthalten sein.
- [−1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Es ist nicht akzeptabel, wenn alle Permutationen von Klausuren generiert und auf Konsistenz geprüft werden.

  Außerdem soll vermieden werden, dass *leichter-als* Relationen zwischen den gleichen Aufgaben mehrmals repräsentiert werden, es sei denn die Handhabung von Konflikten erfordert dies.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Die Anordnung der Aufgaben muss klar erkennbar sein.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens fünf der sechs vorgegebenen Beispiele (`schwierigkeiten0.txt` bis `schwierigkeiten5.txt`) enthalten.

Aufgabe 3: Wandertag

3.1 Lösungsidee

Bei diesem Problem sollen möglichst viele Wunschweglängen erfüllt werden, indem das zugehörige Intervall abgedeckt wird. Das Intervall eines Wunsches ist abgedeckt, wenn mindestens einer der 3 Weglängen in dem Intervall liegt. Ein erster Ansatz zur Lösung dieses Problems besteht darin, alle möglichen Kombinationen von 3 Weglängen zu generieren und zu prüfen, welche Kombination die meisten Intervalle abdeckt. Dabei müssen alle Weglängen zwischen der kleinsten Untergrenze aller Intervalle und der größten Obergrenze aller Intervalle liegen. Wenn A die kleinste untere Grenze und B die kleinste obere Grenze aller Intervalle ist, dann gibt es $B-A+1$ mögliche Weglängen. Diese Menge wird im Folgenden als Kandidatenmenge bezeichnet. Die Anzahl der möglichen Kombinationen aus allen Kandidaten wäre hier $\binom{(B-A+1)}{3}$. In einem Beispiel, in dem sowohl sehr lange als auch sehr kurze Strecken erwünscht sind, kann dies jedoch sehr ineffizient sein.

3.2 Minimierung der Kandidatenmenge

Da für die Lösung des Problems nur die Anzahl der abgedeckten Intervalle relevant ist, müssen unterschiedliche Weglängen, die genau die gleichen Wünsche erfüllen, nicht einzeln betrachtet werden. Im folgenden Beispiel gibt es nur 3 verschiedene Wünsche, die erfüllt werden müssen. Die Intervalle sind: $[1,7]$, $[2,4]$ und $[6,8]$. In der folgenden Tabelle sind alle Intervalle und ihre Bereiche markiert.

Weglänge	1	2	3	4	5	6	7	8
Wunsch 1	[						]	
Wunsch 2		[		]				
Wunsch 3						[		]

Bei diesem Beispiel gibt es 4 interessante Bereiche: In dem weißen Bereich wird nur Wunsch 1 erfüllt. In dem blauen Bereich werden Wunsch 1 und Wunsch 2 erfüllt, in dem roten Bereich die Wünsche 1 und 3 und in dem grauen Bereich nur Wunsch 3. Um die Menge der möglichen Kandidaten für Weglängen zu minimieren, wählen wir für jeden Bereich nur einen Repräsentanten aus. Hierfür ist es ausreichend, lediglich die unteren Grenzen der Intervalle zu betrachten. So kann nun die Menge der möglichen Kandidaten für Weglängen auf die Menge der einzigartigen unteren Grenzen aller Intervalle reduziert werden. Bei diesem Beispiel wird die Kandidatenmenge von $\{1,2,3,4,5,6,7,8\}$ auf $\{1,2,6\}$ reduziert. Für den weißen Bereich wird als Repräsentant die Weglänge 1 gewählt, für den hellblauen Bereich 2 und für den roten Bereich 6. Da der graue Bereich nur eine obere Intervallgrenze enthält, erhält dieser keinen Repräsentanten. Dies ist möglich, weil er keine neuen Wünsche abdeckt, welche nicht schon von einem Kandidaten vor ihm abgedeckt werden. Im allgemeinen reduziert sich die Menge der möglichen Kandidatenweglängen von $(B-A+1)$ auf $\min(B-A+1,n)$, wobei n die Anzahl der Wünsche ist. Das Minimum wird hier verwendet, da es nur maximal so viele Kandidaten geben kann, wie es Ganzzahlen im Bereich $[A,B]$ gibt.

Eine weitere Reduktion ist möglich, wenn die Menge der durch eine Weglänge abgedeckten Intervalle betrachtet werden. So unterscheiden sich die Weglängenkandidaten 1 und 2 nur darin,

Algorithmus 6 GeneriereKandidaten

```
L = Liste der oberen und unteren Grenzen der Intervalle
Sortiere L
K = ∅
for i = 0 to |L| − 2 do
    if L[i] ist untere Grenze und L[i+1] ist obere Grenze then
        Füge L[i] zu K hinzu
    end if
end for
return K
```

dass die Weglänge 2 neben dem Wunsch 1 zusätzlich noch den Wunsch 2 abdeckt. Somit muss die Weglänge 1 nicht betrachtet werden, da die Weglänge 2 bereits alle Wünsche abdeckt, die von Weglänge 1 abgedeckt werden. Die Menge der möglichen Kandidatenweglängen kann also weiter reduziert werden, indem nur Weglängen betrachtet werden, die mindestens einen Wunsch abdecken, der noch nicht von einer anderen Weglänge abgedeckt wird. Im obigen Beispiel wird die Menge der Kandidatenweglängen von 3 auf 2 reduziert. Eine mögliche Kandidatenmenge wäre hier $\{2,6\}$.

3.3 Algorithmus Kandidatenmenge

Um die Kandidatenmenge zu minimieren werden zunächst alle oberen und unteren Grenzen der Intervalle in eine Liste L eingetragen und nach ihren Werten aufsteigend sortiert. Sollten auf eine Weglänge sowohl eine obere als auch eine untere Grenze fallen, so wird die untere Grenze als kleineres Element beim Sortieren betrachtet. Nun wird diese Liste durchlaufen und bei jeder unteren Grenze getestet, ob ihr Nachfolger eine obere Grenze ist. Ist dies der Fall, so wird diese untere Grenze in die Kandidatenmenge aufgenommen. Der Pseudocode in Algorithmus 6 beschreibt dieses Vorgehen.

3.4 Brute Force

Nachdem die Kandidatenmenge generiert wurde, wird nun für jede Kombination von 3 Weglängen getestet, wie viele Intervalle abgedeckt werden. Um herauszufinden, wie viele Intervalle getroffen werden, wird für jedes Intervall getestet, ob es von mindestens einer der 3 gewählten Weglängen abgedeckt wird. Der Pseudocode in Algorithmus 7 gibt einen Überblick über dieses Vorgehen.

3.5 Dynamische Programmierung

Mithilfe von dynamischer Programmierung lässt sich ein effizienterer Ansatz finden. Hierfür wird ein 2-dimensionales Array D erstellt, welches 4 Zeilen und $|K|$ Spalten hat. ($|K|$ ist die Anzahl der Elemente in der Kandidatenmenge). Der eintrag $D[i][j]$ beschreibt die maximale Anzahl erfüllter Wünsche, wenn i Weglängen aus den Kandidaten gewählt wurden und $K[j]$ die größte der gewählten Weglängen ist. Die erste Zeile von D wird mit 0 initialisiert, da mit 0 Weglängen auch keine Wünsche erfüllt werden können. Alle weiteren Zeilen lassen sich aus den vorherigen Zeilen berechnen.

Algorithmus 7 Brute-Force

```
K = GeneriereKandidaten
M = 0                                          ▷ Maximale Anzahl abgedeckter Intervalle
for i = 0 to |K| − 3 do
    for j = i to |K| − 2 do
        for k = j to |K| − 1 do
            m = 0                                  ▷ Anzahl abgedeckter Intervalle
            for Intervall in Intervalle do
                if Intervall wird abgedeckt von K[i] oder K[j] oder K[k] then
                    m = m + 1
                end if
            end for
            if m > M then
                M = m
            end if
        end for
    end for
end for
return M
```

Für das Beispiel mit den Intervallen $[1,7]$, $[2,4]$ und $[6,8]$ aus dem Abschnitt 3.2 sieht das Array D wie folgt aus:

Kandidaten	2	6
0 Weglängen	0	0
1 Weglänge	2	2
2 Weglängen	2	3
3 Weglängen	2	3

Um den Eintrag $D[i][j]$ zu berechnen, wird für alle Werte $k \leq j$ berechnet, wie viel neue Intervalle durch $K[j]$ abgedeckt werden, wenn $K[k]$ bereits gewählt wurde. Die Anzahl der neu abgedeckten Intervalle wird m genannt. $D[i][j]$ wird dann auf den maximalen Wert der Summe $D[i-1][k]+m$ gesetzt. An dem obigen Beispiel ist erkennbar, dass die zweite Zeile genau der Anzahl der abgedeckten Intervalle entspricht, wenn nur die Weglänge $K[j]$ gewählt wird. Der Eintrag $D[2][0]$ ist hier 2, da die maximale Weglänge für diesen Eintrag $K[0] = 2$ ist und somit nur 2 Intervalle abgedeckt werden können. Der Eintrag $D[3][1]$ hingegen ist 3, da die maximale Weglänge für diesen Eintrag $K[1] = 6$ ist und somit die Weglängen 2 und 6 gewählt werden können. Eine dritte Weglänge deckt hier keine weiteren Intervalle ab, da bereits mit 2 Weglängen alle Intervalle abgedeckt wurden. Die maximale Anzahl an abgedeckten Intervallen ist dann der maximale Wert in der letzten Zeile des Arrays D. Der Pseudocode in Algorithmus 8 beschreibt das Berechnen des Arrays D.

3.6 Schnittmengentabelle

Um den dynamischen Ansatz weiter zu optimieren, kann eine Schnittmengentabelle erstellt werden. Hierfür wird ein 2-dimensionales Array S erstellt, welches $|K|$ Spalten und Zeilen hat. Der Eintrag $S[i][j]$ beschreibt die Anzahl der Wünsche, die sowohl von $K[i]$ als auch von $K[j]$

Algorithmus 8 Dynamisch

```
K = GeneriereKandidaten
D = 2-dimensionales Array mit 4 Zeilen und |K| Spalten
Initialisiere D[0][j] = 0 für alle j
for i = 1 to 3 do
    for j = 0 to |K| − 1 do
        m = 0                                   ▷ Maximale Anzahl abgedeckter Intervalle
        for k = 0 to j do
            z = 0                               ▷ Anzahl neu abgedeckter Intervalle
            for intervall in Intervalle do
                if intervall wird abgedeckt von K[j] und nicht abgedeckt von K[k] then
                    z = z + 1
                end if
            end for
            if z + D[i − 1][k] > m then
                m = z + D[i − 1][k]
            end if
        end for
        D[i][j] = m
    end for
end for
```

erfüllt werden. Für das Beispiel mit den Intervallen $[1,7]$, $[2,4]$ und $[6,8]$ aus dem Abschnitt 3.2 sieht das Array S wie folgt aus:

Kandidaten	2	6
2	2	1
6	1	2

In dieser Tabelle lässt sich auch die Anzahl erfüllter Wünsche von einzelnen Weglängen ablesen, indem die Werte $S[i][i]$ betrachtet werden.

Um dieses Array zu berechnen, wird zunächst eine Liste L erstellt, welche alle Intervalle enthält und diese nach ihren unteren Grenzen sortiert. Zudem wird eine im Laufe des Algorithmus eine aufsteigend sortierte Liste E (**E**ndwerte) gehalten, welche die oberen Grenzen aller aktiven Intervalle enthält. Ein Intervall ist aktiv, wenn es sowohl von $K[i]$ als auch von $K[j]$ abgedeckt wird. Der Ablauf des Algorithmus ist in Algorithmus 9 beschrieben.

Mithilfe dieser Tabelle kann nun im Algorithmus 8 die Anzahl z der durch $K[j]$ neu abgedeckten Intervalle effizienter berechnet werden. Statt für jedes Intervall die Anzahl der neu abgedeckten Intervalle z auszuzählen, kann z mit der Formel $z = S[j][j] - S[i][j]$ direkt berechnet werden.

3.7 Laufzeit

Für das Erstellen der Kandidatenmenge muss die Liste der Intervallgrenzen sortiert werden. Diese Liste hat $O(n)$ Elemente, also benötigt das Sortieren $O(n \log n)$ Zeit.

Im Brute-Force Ansatz müssen alle Kombination von 3 Elementen der Kandidatenmenge betrachtet werden und für jede Kombination alle Intervalle durchlaufen werden. Die Laufzeit

Algorithmus 9 Schnittmengentabelle

```
K = GeneriereKandidaten
S = 2-dimensionales Array mit |K| Spalten und Zeilen
L = Liste der Intervalle sortiert nach unteren Grenzen
E = leere Liste
for i = 0 to |L| − 1 do
    entferne Werte aus E, die kleiner als die untere Grenze von L[i] sind
    füge die obere Grenze von L[i] sortiert zu E hinzu
    erstelle T = E
    if L[i] ist in Kandidatenmenge then
        for j = i to |L| − 1 do
            entferne alle Werte aus T, die kleiner als die untere Grenze von L[j] sind
            if L[j] ist in Kandidatenmenge then
                trage |T| in S ein
            end if
        end for
    end if
end for
```

des Brute-Force Ansatzes beträgt somit $O(|K|^3 \cdot n)$, wobei n die Anzahl der Intervalle ist. Zusammen mit dem Aufwand für das Erstellen der Kandidatenmenge beträgt die Laufzeit des Brute-Force Ansatzes $O(n \log n + |K|^3 \cdot n)$.

Beim dynamischen Algorithmus 8 muss für jedes Element des Arrays D die Anzahl der durch die Weglänge $K[j]$ neu abgedeckten Intervalle berechnet werden. Das Berechnen dieser Anzahl benötigt $O(n)$ Zeit, da für jedes Intervall getestet werden muss, ob es durch $K[j]$ abgedeckt wird. Somit beträgt die Laufzeit des dynamischen Algorithmus zusammen mit dem Erstellen der Kandidatenmenge $O(n \log n + |K|^2 \cdot n)$.

Bei der Erstellung der Schnittmengentabelle benötigt das sortierte Einfügen von $L[i]$ in E $O(n)$ Zeit. Dies wird n-mal durchgeführt. Das Entfernen der Werte aus E und T hat einen Aufwand von $O(1)$, da die Listen sortiert sind und somit nur die ersten Elemente der Liste getestet werden müssen. Da diese Entfernen-Operation nur so oft durchgeführt wird, wie es Elemente in L gibt, beträgt die Laufzeit des Algorithmus 9 $O(n^2)$.

Mithilfe dieser Schnittmengentabelle kann die Laufzeit des dynamischen Algorithmus von $O(|K|^2 \cdot n)$ auf $O(|K|^2 + n^2)$ reduziert werden. Der Aufwand für das Erstellen der Schnittmengentabelle ist $O(n^2)$ und die Laufzeit des dynamischen Algorithmus reduziert sich auf $O(|K|^2)$, da das Suchen der neu abgedeckten Intervalle nun in konstanter Zeit möglich ist.

3.8 Speicherbedarf

Der Speicherbedarf des Brute-Force-Ansatzes beträgt $O(|K|)$ um die Kandidatenmenge zu speichern. Der dynamische Algorithmus benötigt $O(|K| + |K|^2)$ Speicher, um das Array D und die Schnittmengentabelle S zu speichern. Sollte die Kandidatenmenge K sehr groß sein, so muss auf die Erweiterung des dynamischen Ansatzes mit der Schnittmengentabelle verzichtet werden.

3.9 Erweiterung

Eine einfache Erweiterung des Problems ist, eine beliebige Anzahl von Weglängen zu wählen. Algorithmisch ändert sich beim dynamischen Ansatz nichts. Für den Brute-Force-Ansatz müssen die drei Schleifen ersetzt werden durch einen Rekursiven Backtracking-Algorithmus. Wenn k Weglängen gewählt werden, ändert sich die Laufzeit des Brute-Force Ansatzes zu $O(|K|^k \cdot n)$. Die Laufzeit des dynamischen Ansatzes ist dann $O(|K|^2 \cdot k + n^2)$. Hier ist zu beachten, dass die Laufzeit des dynamischen Ansatzes linear mit der Anzahl der gewählten Weglängen steigt, die Laufzeit des Brute-Force Algorithmus hingegen exponentiell.

3.10 Beispiele

Die Lösungen für die Beispiele sind nicht einzigartig, da es oft mehrere Weglängen gibt, die genau die gleichen Wünsche abdecken. In der folgenden Tabelle werden ausgewählte Lösungen für die Beispiele gezeigt.

Beispiel	Wünsche ✓	Weglänge 1	Weglänge 2	Weglänge 3
1	6 / 7	22 m	51 m	64 m
2	6 / 6	10 m	60 m	90 m
3	10 / 10	19 m	66 m	92 m
4	79 / 100	524 m	811 m	922 m
5	153 / 200	36696 m	60828 m	88584 m
6	330 / 500	42834 m	74810 m	92920 m
7	551 / 800	39520 m	76088 m	91584 m

Die Wünsche die mit jeder Weglänge erfüllt werden sind in der folgenden Tabelle aufgelistet.

Beispiel 1:

Weglänge	Erfüllte Wünsche	Abgedeckte Intervalle	Index
22	2	$[12,35]$ $[22,45]$	0 1
51	2	$[48,62]$ $[51,57]$	3 4
64	2	$[64,64]$ $[64,71]$	5 6

Beispiel 2:

Weglänge	Erfüllte Wünsche	Abgedeckte Intervalle	Index
10	2	$[10,30]$ $[10,50]$	4 5
60	3	$[60,80]$ $[40,80]$ $[40,60]$	0 2 3
90	1	$[90,90]$	1

Beispiel 3:

Weglänge	Erfüllte Wünsche	Abgedeckte Intervalle	Index
19	3	$[16,65]$ $[17,27]$ $[19,22]$	6 8 9
66	4	$[53,90]$ $[27,67]$ $[26,72]$ $[66,67]$	0 1 2 5
92	3	$[92,94]$ $[89,94]$ $[90,96]$	3 4 7

Beispiel 4:

Weglänge	Erfüllte Wünsche	Abgedeckte Intervalle
524	38	3 6 7 9 10 11 12 19 20 23 24 25 34 37 38 39 40 42 47 54 56 57 58 60 64 65 67 71 78 80 81 82 83 90 91 97 98 99
811	37	0 1 3 4 5 8 13 14 16 18 19 20 21 25 26 27 29 31 32 34 43 48 51 53 55 59 62 64 71 73 74 75 82 87 88 96 97
922	20	3 18 22 30 34 35 36 46 48 50 51 62 66 69 72 75 86 89 92 94

Für alle weiteren Beispiele werden nur noch die (0-basierten) Indizes der erfüllten Wünsche angegeben. Die Reihenfolge ist hierbei die gleiche wie in den Beispieldateien.

Beispiel 5:

Weglänge	Erfüllte Wünsche	Indizes der Wünsche
36696	86	1 2 5 7 11 14 17 20 21 26 27 29 32 37 38 39 40 42 44 45 47 54 63 64 66 67 71 72 73 78 86 88 89 91 93 94 95 96 97 98 102 103 104 106 107 108 110 111 113 114 115 117 118 121 123 124 126 127 128 133 135 136 144 145 151 161 163 168 170 174 175 176 177 178 180 181 182 186 189 190 191 192 194 195 197 199
60828	86	1 4 5 10 11 13 20 21 24 26 29 31 32 33 34 37 39 40 41 42 45 48 52 54 57 62 63 64 65 72 81 84 85 87 91 94 96 101 102 104 105 107 108 111 113 114 117 118 124 128 129 130 131 132 133 135 137 138 143 144 145 147 149 151 154 157 158 161 162 166 168 170 171 173 175 177 178 182 184 186 191 192 193 194 195 196
88584	52	3 5 6 8 9 10 13 15 16 20 22 26 29 30 33 34 39 53 56 59 76 77 80 84 90 94 100 105 114 116 119 120 130 131 139 140 142 144 147 148 150 151 153 157 165 169 170 172 175 184 187 191

Beispiel 6:

Weglänge	Erfüllte Wünsche	Indizes der Wünsche
42834	168	0 9 18 25 30 32 33 40 41 43 44 45 46 50 51 52 53 54 56 58 60 64 69 70 71 74 75 81 85 86 87 88 94 99 100 102 110 112 113 114 117 120 121 125 128 129 132 136 137 144 145 148 149 150 152 157 163 165 166 174 175 179 181 184 185 189 193 194 195 196 197 202 203 205 210 211 214 215 221 223 228 235 236 237 241 244 248 249 251 254 255 257 260 261 262 266 276 277 278 279 280 283 284 288 290 293 297 299 300 301 306 307 311 312 314 315 318 319 324 325 328 341 344 346 348 350 353 358 360 362 366 367 368 373 377 379 382 384 385 390 393 396 399 404 405 412 414 415 417 418 429 431 432 436 447 451 456 457 460 463 469 473 477 478 481 487 490 496
74810	167	2 4 7 11 12 19 25 27 29 32 33 34 35 37 39 42 46 47 51 54 55 59 67 70 71 79 81 83 87 90 93 98 102 103 106 107 108 110 111 114 115 117 120 128 129 131 132 135 140 141 142 143 145 146 148 151 152 157 164 165 174 175 176 178 179 184 186 188 194 195 197 198 200 206 213 214 215 216 221 225 229 230 234 243 244 249 254 259 268 277 284 287 291 292 295 297 298 299 301 306 309 311 316 318 320 322 326 329 330 331 334 341 342 343 346 347 349 351 352 354 366 367 368 371 374 379 381 383 385 386 389 391 397 398 403 404 409 412 416 419 428 429 430 432 435 438 442 443 444 447 449 451 453 456 458 468 469 470 474 476 477 478 480 487 490 494 496
92920	102	6 7 11 12 20 27 31 32 46 47 49 63 70 79 84 91 92 93 97 106 115 123 126 132 138 141 142 143 154 157 168 171 177 179 180 182 184 190 195 197 198 200 206 215 224 226 231 232 238 247 253 258 264 268 272 275 289 295 297 299 305 306 322 334 335 336 342 345 346 349 356 363 369 370 381 383 387 388 394 395 409 410 420 425 426 433 435 448 450 451 454 462 465 467 469 470 479 480 482 484 486 497

Beispiel 7:

Weglänge	Erfüllte Wünsche	Indizes der Wünsche
39520	285	3 4 5 8 9 10 11 15 16 21 26 27 28 29 30 37 40 41 42 44 45 58 60 61 63 65 69 70 75 76 78 82 85 87 88 90 94 95 98 103 104 106 110 114 115 117 123 126 132 134 136 141 147 148 152 154 157 159 160 161 162 163 165 169 170 173 174 175 179 180 182 184 190 191 195 207 209 210 212 214 215 218 219 224 228 229 237 238 244 245 251 252 254 261 265 269 272 276 283 284 286 287 291 292 294 298 301 303 304 307 308 309 311 313 319 320 321 325 326 331 336 340 342 347 351 353 356 357 360 363 365 370 372 374 375 379 380 381 382 384 386 390 393 396 398 400 403 407 409 412 414 421 423 425 433 436 441 444 446 448 449 453 462 464 466 469 471 475 476 478 485 488 493 496 498 500 502 504 505 509 511 512 517 519 520 523 524 527 528 529 530 531 532 533 539 541 545 547 550 558 559 566 569 573 576 577 581 582 583 584 588 594 596 601 603 611 613 615 617 619 622 623 627 630 631 633 635 637 638 640 643 648 649 651 658 663 665 666 668 671 672 675 678 681 685 688 700 704 705 707 708 710 712 723 725 727 728 730 732 733 736 739 741 742 747 751 755 756 757 760 762 766 767 773 775 778 780 782 786 789 790 794 795 797 799
76088	278	0 6 8 10 11 12 15 20 22 25 27 32 34 37 41 46 49 54 57 58 61 62 63 64 66 68 69 73 79 80 85 86 91 92 94 98 100 109 110 111 113 115 119 121 122 124 125 126 128 133 136 138 141 145 147 152 157 166 169 170 174 179 183 184 185 186 187 188 192 194 195 197 201 202 204 208 210 216 217 219 222 224 228 229 231 233 237 243 244 248 249 251 255 256 257 258 267 270 276 277 280 284 289 290 295 298 300 304 305 307 311 312 315 317 319 325 334 337 339 342 343 346 348 349 352 358 359 360 366 368 370 373 375 379 383 385 388 394 395 403 409 417 420 422 424 425 426 427 428 434 440 449 454 455 456 460 468 469 472 475 478 479 480 481 487 488 491 494 496 498 505 506 507 532 534 536 537 538 539 541 544 549 550 551 555 561 562 563 565 566 569 572 573 577 581 582 583 584 586 588 589 590 591 592 594 600 604 605 607 609 610 611 621 622 624 628 631 634 636 638 640 644 645 646 651 657 660 663 665 671 672 677 687 690 691 695 696 698 699 700 704 706 710 711 712 715 716 718 725 727 729 732 735 737 740 741 743 746 749 755 756 757 759 762 763 764 765 766 767 770 772 776 782 784 785 790 794 798
91584	195	11 12 14 19 27 37 43 46 53 54 55 56 57 59 61 62 64 66 68 72 74 79 80 81 92 94 100 101 102 109 111 113 121 125 130 135 141 146 150 152 153 164 167 169 171 172 176 184 185 187 188 192 194 195 197 198 204 213 217 224 228 231 233 237 243 248 249 251 255 257 259 270 275 276 277 282 314 318 334 342 343 360 362 368 369 370 375 377 379 387 391 395 397 401 405 406 411 413 415 416 417 418 419 425 429 432 439 452 456 467 469 475 478 479 480 487 488 489 492 498 505 508 510 514 521 526 538 540 541 546 548 549 550 551 554 556 557 561 565 566 569 574 580 581 582 584 587 592 605 606 607 610 620 626 629 632 636 644 646 647 653 654 657 664 674 677 682 686 695 697 698 704 711 713 718 719 729 741 744 748 754 757 758 761 764 765 776 779 781 782 784 787 790 792 794

3.11 Bewertungskriterien

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

- [−1] **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**

- [−1] **Modellierung ungeeignet**
  Die Wunsch-Intervalle müssen als inklusive modelliert werden.
- [−1] **Lösungsverfahren fehlerhaft**
  Die Teilnahmezahl muss korrekt sein. Insbesondere darf jede Person nur für eine Weglänge innerhalb ihres Wunschintervalls gezählt werden. Das Verfahren soll außerdem die optimale Lösung finden.
- [−1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Ein Brute-Force-Algorithmus, der alle möglichen Kombinationen testet, ist nur dann zulässig, wenn die Anzahl der möglichen Kandidaten für Weglängen vorab reduziert wurde: Es dürfen maximal so viele Kandidatenweglängen betrachtet werden, wie es Intervalle gibt.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Für jedes dokumentierte Beispiel muss angegeben sein
  - die Gesamtanzahl der teilnehmenden Personen,
  - die drei berechneten Weglängen und
  - eine Zuordnung der Personen zu den Weglängen.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens fünf der sieben vorgegebenen Beispiele (`wandern1.txt` bis `wandern7.txt`) enthalten. Mindestens eines der größeren Beispiele `wandern6.txt` oder `wandern7.txt` muss enthalten sein.

  Die Tabelle gibt die optimalen Ergebnisse an. Dabei ist jede Kombination aus 3 Weglängen, die jeweils in genau einem der genannten inklusiven Intervalle liegen, optimal.

Beispiel	Wünsche ✓	Intervall 1	Intervall 2	Intervall 3
1	6 / 7	[22, 35]	[51 ,57]	64
2	6 / 6	[10, 30]	60	90
3	10 / 10	[19, 22]	[66, 67]	[92, 94]
4	79 / 100	[524, 542]	[811, 812]	[922, 928]
5	153 / 200	[36696, 36728]	[60828, 61602]	[88584, 88801]
6	330 / 500	[42834 ,42898]	[74810 ,74869]	[92920, 93177]
7	551 / 800	[39520, 39543]	[76088 ,76136]	[91584, 91601]

Aufgabe 4: Krocket

4.1 Lösungsidee

In dieser Aufgabe sind die Positionen und die Reihenfolge der Tore eines Krocketspiels gegeben. Es soll herausgefunden werden, ob es möglich ist, einen Ball mit Radius r mit einem einzigen Schlag in der richtigen Reihenfolge durch alle Tore zu befördern. Dies führt zu einer geometrischen Fragestellung:

Gegeben sind n Strecken in der Ebene, die sich paarweise nicht schneiden und die durch die Koordinaten ihrer Endpunkte beschrieben werden. Die Strecken repräsentieren die Tore, ihre Endpunkte die Positionen der dazugehörigen Torpfosten. Es muss bestimmt werden, ob eine Gerade g mit den folgenden drei Eigenschaften existiert:

1. Jede der Torstrecken hat einen Schnittpunkt mit g.
2. Entlang der Geraden g (d.h. in einer von zwei möglichen Richtungen) haben diese Schnittpunke die richtige Reihenfolge.
3. Jeder der Torendpunkte hat einen Abstand von r zu g.

Eine Gerade mit diesen drei Eigenschaften soll im Folgenden als *nützlich* bezeichnet werden. Wenn es eine nützliche Gerade gibt, kann sich das Zentrum des Balls auf dieser bewegen und durchquert dabei alle Tore in der richtigen Reihenfolge, ohne mit den Torpfosten zu kollidieren. Gibt es keine nützliche Gerade, reicht ein einziger Schlag nicht aus, um den Parcour zu absolvieren. Dabei wird angenommen, dass die Torpfosten Radius 0 haben und der Ball somit nur mit ihnen kollidiert, wenn die Distanz zwischen Pfosten und Zentrum des Balls echt kleiner als r ist. Das stimmt aber nur, wenn der Ball die Tore *vollständig* durchqueren muss. Andernfalls, also wenn der Ball beispielsweise *im* ersten Tor starten dürfte, könnte der Abstand der Geraden zu den Endpunkten der ersten bzw. letzten Tore auch geringer sein.

Nun sollen alle Geraden in der Ebene überprüft werden. Ist eine von ihnen nützlich, kann auf ihr ein Punkt weit entfernt von der Punktmenge gewählt werden. Von dort aus wird der Ball dann entlang der Geraden abgeschlagen. Problematisch dabei ist nur, dass es in der Ebene unendlich viele Geraden gibt, die selbstverständlich nicht alle durchgegangen werden können. Die eigentliche Schwierigkeit der Aufgabe besteht also darin, aus der unendlich großen Menge aller Geraden eine endliche Menge von Kandidaten auszuwählen. Diese muss sicher ausreichen, um eine eventuell vorhandene Lösung zu finden. Dafür lässt sich eine hilfreiche Beobachtung machen:

Wenn es eine nützliche Gerade gibt, gibt es auch eine nützliche Gerade, die von mindestens zwei Torendpunkten einen Abstand von genau r hat.

Gibt es nämlich eine nützliche Gerade, so muss jede der durch sie begrenzten Halbebenen aufgrund der ersten Eigenschaft die Hälfte der Torendpunkte enthalten. Sie kann also parallel in eine beliebige Richtung verschoben werden, bis sie von mindestens einem Torendpunkt P einen Abstand von genau r hat. Der Kreis mit Radius r um den Punkt P wird dann von dieser Geraden berührt. Sie kann also „um diesen Kreis“ gedreht werden, bis sie von mindestens einem weiteren Torendpunkt einen Abstand von genau r hat. Wenn die Gerade am Anfang nützlich war, ist sie das nach diesen beiden Operationen immer noch. Nun hat sie aber die gewünschte Eigenschaft.

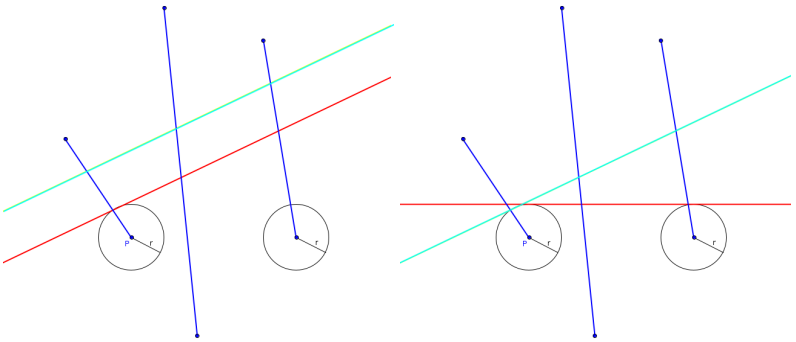


Abbildung 4.1: Die beiden Schritte zum Verändern einer nützlichen Gerade: Links das Verschieben der Gerade und rechts das Drehen um den Kreis um P. Die Ausgangsposition ist jeweils türkis und die Zielposition jeweils rot dargestellt. Die Tore sind blau.

Damit kann die Aufgabe gelöst werden: Es werden alle Paare von Torendpunkten durchgegangen. Jedes von diesen Paaren ergibt vier Geraden als Kandidaten: Für jeden der beiden Punkte gibt es nämlich zwei Möglichkeiten, auf welcher Seite der Geraden er liegen soll. Sind aber diese beiden Wahlen getroffen, so ist die Gerade mit Abstand r zu beiden Punkten eindeutig festgelegt. Nun muss noch für alle Kandidaten überprüft werden, ob sie nützlich sind.

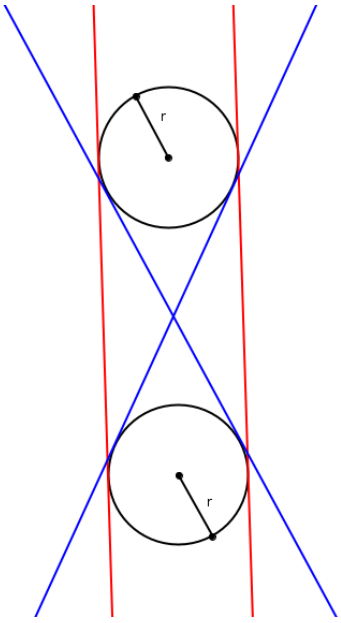


Abbildung 4.2: Die vier zu einem Punktepaar gehörenden Kandidaten. Die roten berühren die Kreise auf der gleichen Seite, die blauen auf unterschiedlichen Seiten.

Zu diesem Zweck muss für jeden Torendunkt überprüft werden, ob sein Abstand zur Geraden ausreichend groß ist. Außerdem muss für jede Torstrecke ihr Schnittpunkt mit der Geraden berechnet werden bzw. getestet werden, ob dieser überhaupt existiert. Abschließend wird noch

die Reihenfolge der Schnittpunkte entlang der Geraden betrachtet. Diese ist genau dann korrekt, wenn die Folgen ihrer x- und y-Koordinaten monoton sind. Wenn sich das Zentrum des Balls auf der Geraden bewegt, verändern sich die Koordinaten dieses Punktes schließlich ebenfalls monoton.

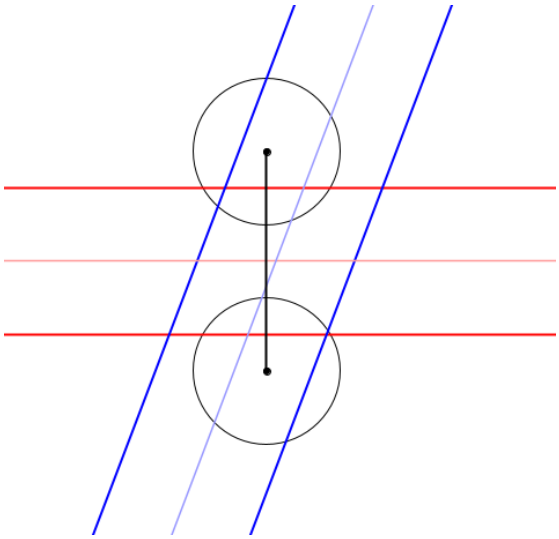
4.2 Laufzeit

Der für die Laufzeit ungünstigste Fall tritt ein, wenn es keine nützliche Gerade gibt. Dann werden für jedes Paar von Torendunkten vier Geraden getestet. Das sind für n Tore genau $4 \cdot \binom{2n}{2} = 4n(2n-1)$, also $O(n^2)$ viele. Für jede dieser Geraden werden im schlimmsten Fall alle Torstrecken und -endunkte getestet, wenn nur im jeweils letzten Test festgestellt wird, dass die Gerade nicht nützlich sein kann.
Die Anzahl der Punkte und Strecken liegt in $O(n)$, wobei die Bestimmung eines einzelnen Schnittpunkts oder Abstands in konstanter Laufzeit geschehen kann. Die Überprüfung auf Monotonie der Koordinatenfolgen der Schnittpunkte lässt sich in linearer Laufzeit umsetzen. Insgesamt liegt die Laufzeitkomplexität des oben beschriebenen Algorithmus also in $O(n^3)$.
In der Praxis sind die Laufzeiten des Algorithmus aber deutlich geringer, schließlich kann die Überprüfung aller Endpunkte und Strecken abgebrochen werden, sobald eine die Eigenschaften einer nützlichen Geraden verletzt.

4.3 Fallstricke

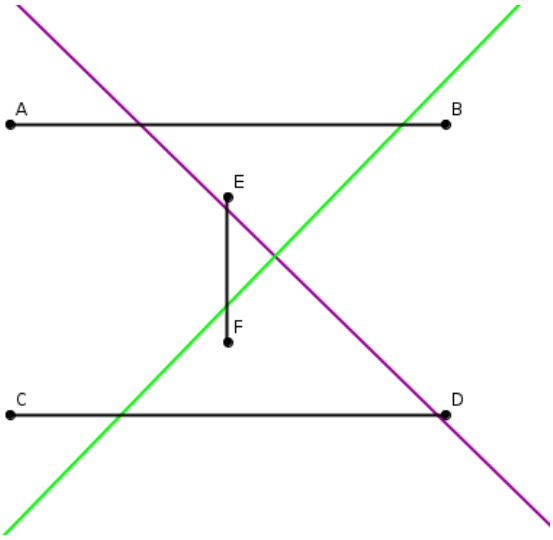
Diese Aufgabe hat einige Lösungsansätze, die auf den ersten Blick sehr plausibel wirken, sich aber trotzdem als falsch herausstellen. Zwei davon sollen hier kurz vorgestellt und widerlegt werden.

Es scheint zunächst, als könne man eine Lösung für $r = 0$ auf den allgemeinen Fall durch Kürzen der Tore um r an beiden Enden zurückführen. Wenn eine Gerade die so verkürzte Torstrecke schneidet, hat der Schnittpunkt vom eigentlichen Torenndpunkt der Strecke schließlich einen Abstand von mindestens r. Dann müsste für eine gegebene Menge von Torstrecken lediglich eine Gerade gefunden werden, die alle Strecken in der richtigen Reihenfolge schneidet. Wenn sich das Zentrum des Balls auf einer Geraden bewegt, ist der minimale Abstand zwischen dem Zentrum und einem Torendpunkt aber nicht zwingend der Abstand zwischen diesem Endpunkt und dem Schnittpunkt von Torstrecke und Gerade, wie die folgende Abbildung verdeutlicht:



Dargestellt ist eine Torstrecke. In diesem Beispiel soll $r = 1$ sein; die beiden Kreise um die Endpunkte haben genau diesen Radius. Zwischen je zwei gleichfarbigen Geraden kann sich dann ein Ball mit Radius r bewegen, sein Zentrum befindet sich dabei immer auf der etwas helleren Mittellinie. Im Fall der roten Parallelen passt er auch durch das Tor, im Fall der blauen aber nicht – obwohl die Gerade, auf der sich sein Zentrum bewegt, die schwarze Strecke in einem Punkt schneidet, der von beiden Endpunkten einen Abstand von mehr als 1 hat. Die hellblaue Gerade schneidet nämlich die beiden Kreise und ist somit nicht nützlich.

Es lässt sich auch beobachten, dass eine nützliche Gerade die Menge aller Torendpunkte in zwei Hälften zerlegt, wobei jedes Tor einen Endpunkt auf jeder Seite der nützlichen Geraden hat. Entpsrechend ließe sich auch versuchen, die Endpunkte erst in zwei Hälften aufzuteilen und dann eine Gerade „zwischen“ diesen Punktmengen zu finden. Die folgende Abbildung belegt aber, dass die Aufteilung der Punktmenge im Allgemeinen nicht eindeutig ist:



Beide Geraden schneiden alle Torstrecken, teilen die Punktmenge aber unterschiedlich auf: Die grüne führt zur Aufteilung in $\{A,C,E\}$ und $\{B,D,F\}$, während die lilane die beiden Mengen

$\{A,C,F\}$ und $\{B,D,E\}$ erzeugt. Die Abbildung belegt auch, dass die Eckpunkte zweier Tore nicht zwingend ein konvexes Viereck bilden müssen - so liegt der Punkt E etwa im Dreieck $\triangle BAF$.

4.4 Optimierung

Der hier vorgestellte Algorithmus hat eine Laufzeitkomlexität von $O(n^3)$. Es ist aber auch möglich, das Problem in Laufzeitkomplexität $O(n^2 \log n)$ zu lösen. Dafür sind fortgeschrittenere Konzepte nötig, weshalb diese Lösung hier nur kurz skizziert und ihre Laufzeitkomplexität auch nicht von den Algorithmen der Teilnehmenden erwartet wird:

Ein für geometrische Probleme oft hilfreiches Paradigma ist das sogenannte *Sweep-Verfahren*[13]. In diesem Fall kann für einen konkreten Torendpunkt Q eine Variation des *radial sweep* ausgeführt werden, um nützliche Geraden mit einem Abstand von r zu Q zu finden: Dabei werden zunächst alle anderen Torendpunkte nach ihrem Winkel zu Q sortiert. Dann wird eine Gerade mit Abstand r zu Q „um“ den Kreis mit Radius r um Q gedreht, wobei sie immer anhält, wenn sie von einem der anderen Punkte einen Abstand von genau r hat. Wenn die anderen Punkte und die Torstrecken bei diesem Sweep intelligent verwaltet werden, kann die Gerade in $O(n)$ einmal um diesen Kreis gedreht und bei jedem Anhalten überprüft werden, ob die drei Bedinungen für ihre Nützlichkeit erfüllt sind. Weil das initiale Sortieren $O(n \log n)$ in Anspruch nimmt und der Sweep im schlimmsten Fall für jeden Endpunkt durchgeführt werden muss, ergibt sich eine Laufzeitkomplexität von $O(n^2 \log n)$.

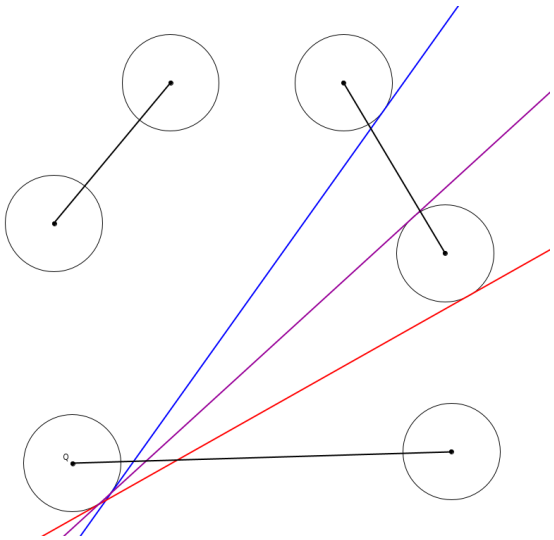


Abbildung 4.3: In diesem Beispiel könnte beispielsweise eine Gerade von der roten über die lilane zur blauen Position gedreht werden.

4.5 Beispiele

Die Aufgabenstellung fordert die Ausgabe des Startpunkts des Balls und seiner Schlagrichtung, also eines Bewegungsvektors. Es sind aber auch Alternativen denkbar, etwa die Ausgabe von

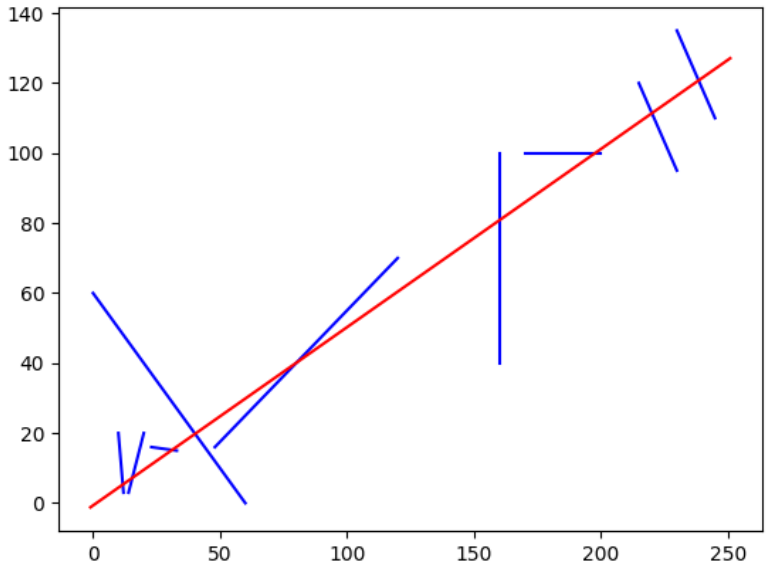
[13] Siehe beispielsweise `https://de.wikipedia.org/wiki/Sweep_(Informatik)`

Start- und Endpunkt oder eine Kombination aus Geradengleichung und Startpunkt und Richtung entlang dieser Geraden.

In den folgenden Abbildungen sind die Tore des Krocketspiels als blaue Linien und die Strecke, auf der sich der Schwerpunkt des Balls bewegt, als rote Linie dargestellt.

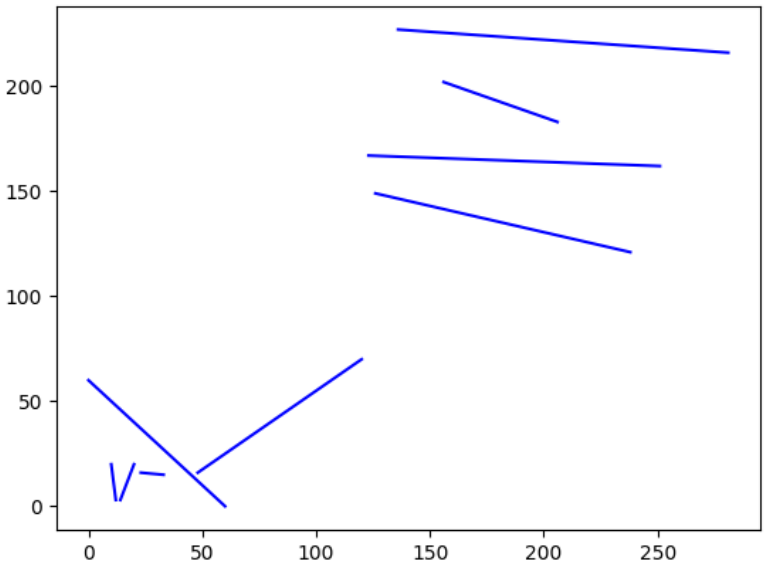
krocket1.txt

Im ersten Beispiel kann der Parcours mit einem einzigen Schlag absolviert werden, beispielsweise durch Abschlagen vom Punkt $(-1000, -509.656)$ in Richtung $(61000, 31047.9)$.



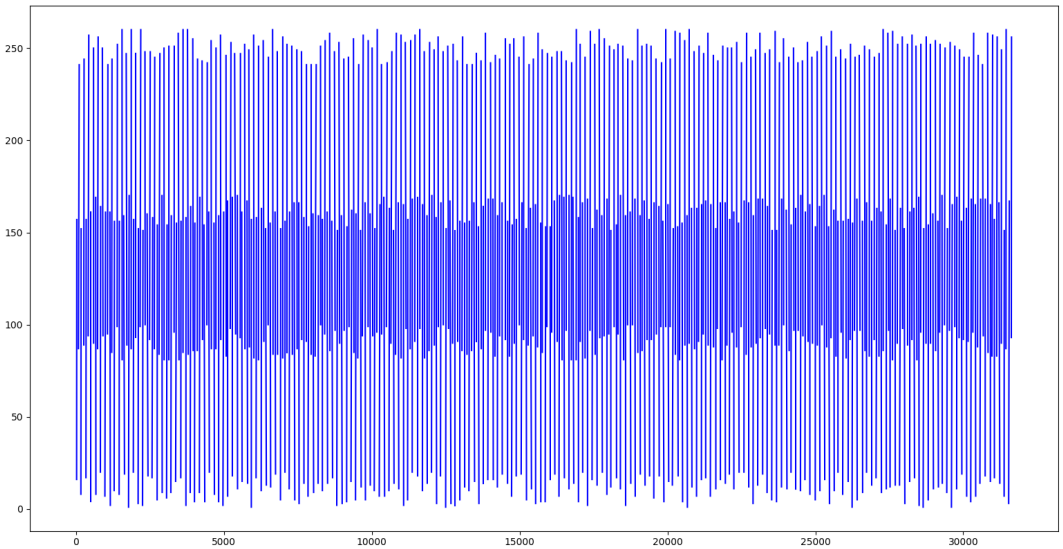
krocket2.txt

Die zweite Beispieleingabe besitzt keine Lösung, weil es nicht möglich ist, den Ball mit einem Schlag durch alle Tore zu befördern.



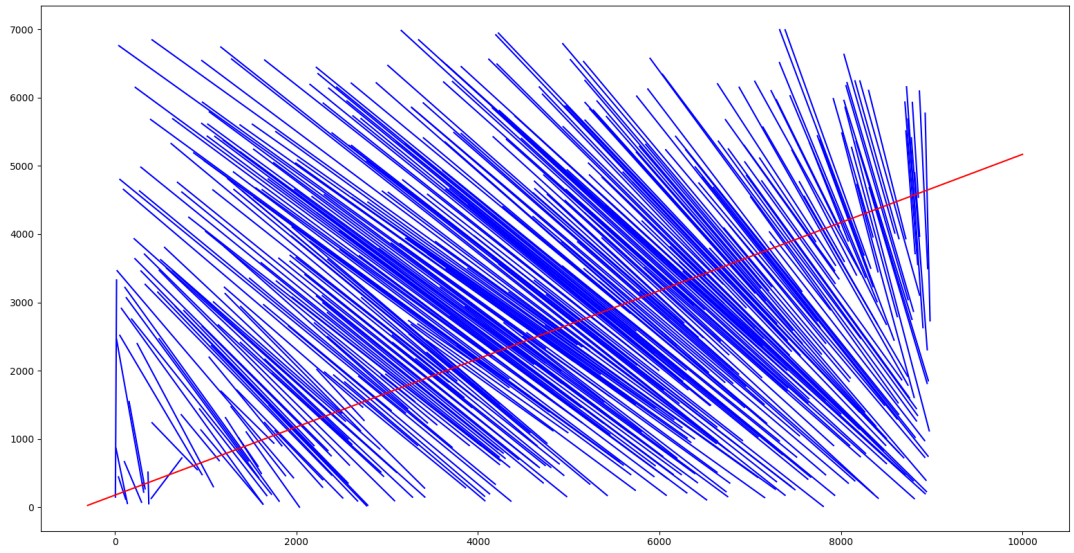
krocket3.txt

Die dritte Beispieleingabe besitzt keine Lösung, obwohl es möglich ist, den Ball mit einem Schlag durch alle Tore zu befördern. Dabei lässt sich die richtige Reihenfolge nämlich nicht einhalten.



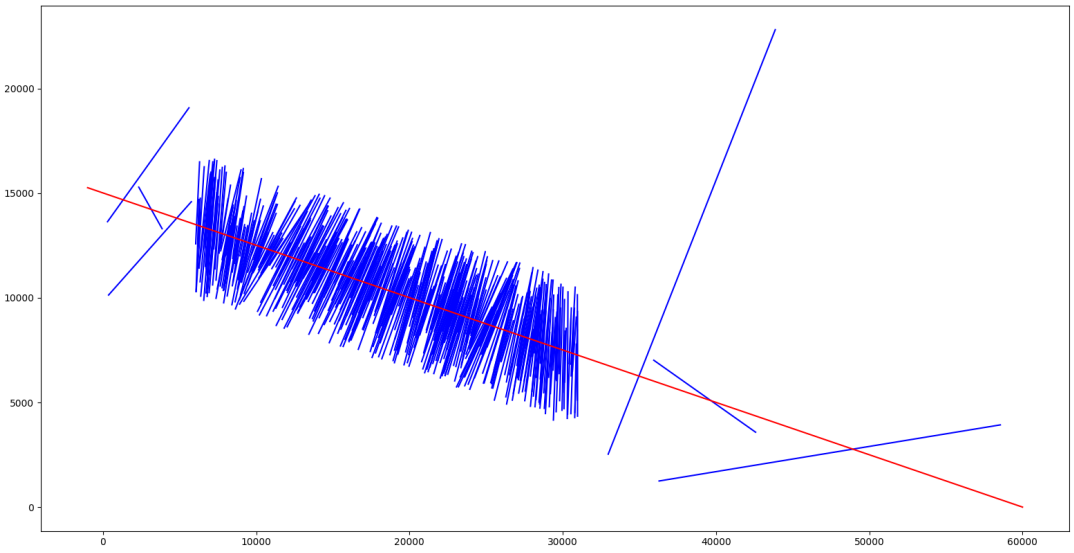
krocket4.txt

Das vierte Beispiel hat wieder eine Lösung, für die etwa vom Punkt $(-1000, -323.2)$ in Richtung $(61000, 30465.4)$ geschlagen werden kann.



krocket5.txt

Für den letzten Parcours kann sich der Ball zum Beispiel vom Punkt $(-1000, 15263.1)$ in Richtung $(61000, -15261.6)$ bewegen.



4.6 Bewertungskriterien

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

- $[-1]$ **Nur Radius $r = 0$ betrachtet**
  Für einen Lösungsansatz ist es in Ordnung $r = 0$ anzunehmen, für die tatsächliche Lösung allerdings nicht.
- $[-1]$ **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
- $[-1]$ **Modellierung ungeeignet**
  Die geometrischen Strategien müssen korrekt gewählt werden. Es muss beachtet werden, dass der Ball für die gesamte Strecke mit seinem gesamten Radius innerhalb der Tore liegt. Es darf nur eine einzelne Gerade verwendet werden, um alle Tore zu durchqueren. Es ist akzeptabel, wenn der Ball im ersten Tor startet.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Die genutzten geometrischen Strategien müssen korrekt umgesetzt sein. Alle Tore müssen in der richtigen Reihenfolge durchquert werden. Die Art des Umgangs mit Kommazahlen darf nicht zu Fehlern bei den Ergebnissen führen; insbesondere muss auf korrekte Vergleiche geachtet werden. Der Ball darf bspw. nicht durch Rundungsfehler doch außerhalb eines Tores landen.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Die Laufzeit des Verfahrens sollte polynomiell in der Anzahl der Tore sein. Außerdem darf das Lösungsverfahren nicht lediglich ohne Strategie möglichst viele Schläge (Geraden) durchprobieren.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Es müssen ein Startpunkt und eine Schlagrichtung ausgegeben werden. Eine graphische Darstellung ist sehr hilfreich, ist aber in der Aufgabenstellung nicht gefordert und wird deshalb nicht erwartet.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu allen fünf vorgegebenen Beispielen (`krocket1.txt` bis `krocket5.txt`) enthalten.

Aufgabe 5: Das ägyptische Grabmal

5.1 Lösungsidee

Um ein Programm zu schreiben, welches Petra sicher zum ägyptischen Grabmal leitet, wollen wir zunächst einige allgemeine Überlegungen zur Aufgabe anstellen. Dabei ist die wahrscheinlich drängendste Frage, ob es immer einen Weg für Petra durch das Labyrinth[14] geben muss, oder ob es auch Rätsel geben kann, in denen sie ewig gefangen wäre.

Um diese Frage zu klären, wollen wir den Größen im Rätsel zunächst genaue Namen geben. Die Perioden P sind ein Tupel $P = (p_1, \ldots, p_n)$ der jeweiligen Quaderperioden p_i (in Minuten) an der Stelle $i = 1, \ldots, n$ bei insgesamt n Quadern im Labyrinth. Ob ein jeweiliger Abschnitt offen oder geschlossen ist, kann durch die Modulooperation[15] aus der aktuellen Zeit t (beginnend bei $t = 0$ und ebenfalls in Minuten) geschlossen werden. Für jeden Abschnitt i gilt, dass er alle $2p_i$ Zeitschritte schließt und alle $2p_i + p_i$ Zeitschritte öffnet. Der aktuelle Status eines Abschnitts kann also durch die folgende Funktion beschrieben werden

$$\textsc{Status}(\text{Abschnitt } i \text{ zur Zeit } t) \equiv \begin{cases} \textsc{geschlossen}, & \text{wenn } t \bmod 2p_i < p_i, \\ \textsc{offen}, & \text{wenn } t \bmod 2p_i \geq p_i. \end{cases} \tag{5.1}$$

Nun betrachten wir die Zeit $T(P)$, die sich aus dem doppelten Produkt aller Perioden ergibt, also

$$T(P) = 2\prod_i p_i.$$

Zu dieser Zeit sind alle Steine unten, weil für jeden Abschnitt die Modulobedingung in Gleichung (5.1) erfüllt ist; einen Zeitschritt davor sind jedoch alle Steine oben[16] und damit der Weg durch das Grabmal für Petra vollkommen frei. Um dies zu beweisen, betrachten wir für das Problem P einen beliebigen Abschnitt j mit Periode p_j und berechnen

$$T(P) - 1 \pmod{2p_j} = 2p_j\Big(\prod_{i \neq j} p_i\Big) - 1 \pmod{2p_j} = 2p_j - 1 \pmod{2p_j}.$$

Dies ist klar größer-gleich p_j, mit Gleichheit bei $p_j = 1$, und damit gilt also für alle Abschnitte j, dass $\textsc{Status}(\text{Abschnitt } j \text{ zur Zeit } T(P) - 1) = \textsc{offen}$[17]. Spätestens dann kann Petra also ungestört durch das Labyrinth zum Grabmal spazieren. Es gibt also immer eine Lösung und auch immer eine Lösung, bei der gleichzeitig alle Abschnitte geöffnet sind und die Anweisungen an Petra lediglich lauten: *Warte $T(P)$ Minuten, laufe zum Grabmal*. Diese Lösung stellt jedoch nicht den schnellsten Weg zum Grabmal dar.

[14] Hier und im Folgenden wird der Gang als Labyrinth bezeichnet.

[15] Für zwei natürliche Zahlen a und b definiert $a \bmod b$ den Rest, der bei Division von a durch b bleibt. Ist $a = nb + m$ für ebenfalls natürliche Zahlen n und m, so ist $a \bmod b = m$. Wählen wir beispielsweise $a = 17$ und $b = 3$, so ist $a = 5 \times 3 + 2$, und dementsprechend $17 \bmod 3 = 2$. Für die Notation der Aufgabe ist es wichtig, dass in unserer Definition $mb \bmod b = 0$, also beispielsweise $15 \bmod 3 = 0$, ist.

[16] Diesen Umstand können wir uns auch an einem Beispiel vergegenwärtigen. Betrachten wir dazu das sehr einfache Grabmal mit den beiden Perioden $p_1 = 2$ und $p_2 = 3$. Zur Zeit $t = T(P) = 2 \times (2 \times 3) = 12$ sind alle Steine gerade unten und das Rätsel befindet sich im selben Zustand wie zur Zeit $t = 0$. Gehen wir einen Zeitschritt zurück zu $t = T(P) - 1$, müssen entsprechend alle Steine oben sein.

[17] Technisch reicht an dieser Stelle das kleinste gemeinsame Vielfache aller Zahlen, also $T(P) = 2\text{kgV}(p_1, \ldots p_n)$, das Argument bleibt das gleiche.

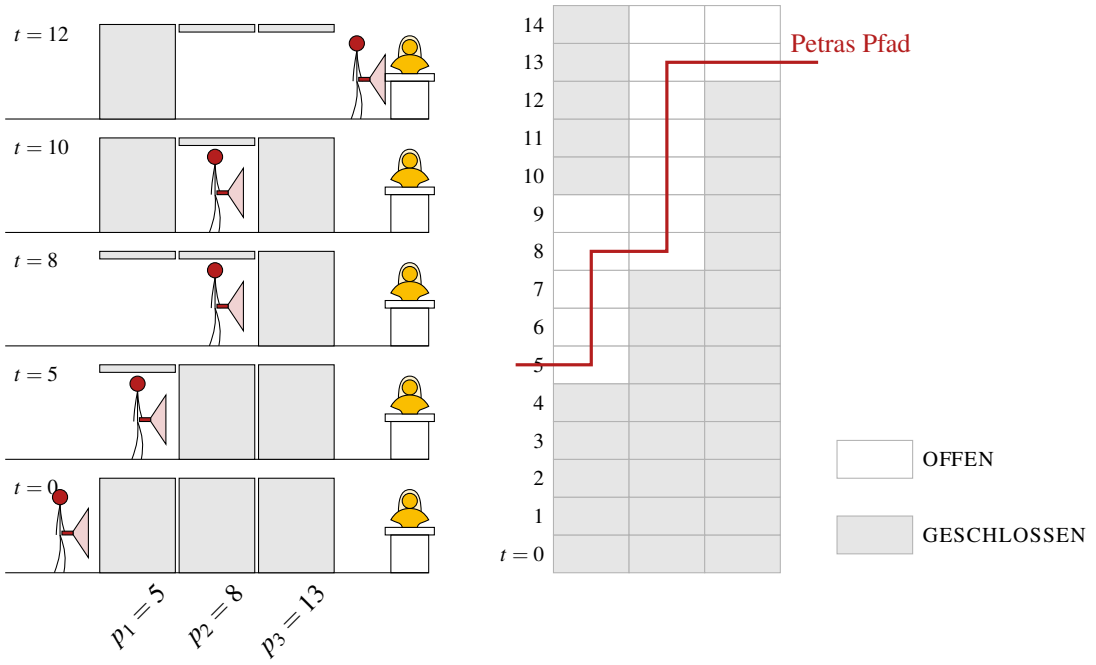


Abbildung 5.1: Links: Darstellung der verschiedenen Konfigurationen des Labyrinths aus dem Aufgabenblatt $P = (5, 8, 13)$ mit den jeweiligen Positionen von Petra, die den schnellstmöglichen Weg zum Grabmal ermöglichen. Rechts: Die Übersetzung des gleichen Labyrinths in ein Zustandsraster, bei dem eine graue Zelle für Abschnitt i und Zeit t darstellt, dass eben dieser Abschnitt i zur Zeit t nicht zugänglich ist. Eine weiße Zelle stellt entsprechend einen zugänglichen Abschnitt dar.

Um nun konkrete Lösungsansätze zur Bestimmung des schnellstmöglichen Wegs durch das Labyrinth zu finden, müssen noch einige offene Details geklärt werden.

1. Wir nehmen an, dass Petra keine Bewegungszeit von einem Abschnitt zum nächsten benötigt. So kann sie innerhalb der einen Minute in einen beliebigen (erreichbaren) Abschnitt einer Konfiguration laufen.
2. Ebenfalls kann Petra Abschnitte zurücklaufen, also beispielsweise von Abschnitt 5 zu Abschnitt 2 gehen, insofern sich dadurch das Grabmal schneller erreichen lässt.
3. Den schnellstmöglichen Weg definieren wir zunächst lediglich durch die Zeit, nach der das Grabmal erreicht werden kann, und nicht über zusätzliche Metriken, wie etwa die Komplexität der Anweisungen oder der insgesamt zurückgelegten Strecke.

Darstellung des Problems

Um dem Problem Herr zu werden, benötigen wir eine geeignete Darstellung des Problems, auf der unsere Algorithmen arbeiten können. Dafür ordnen wir die Konfigurationen analog zur Illustration in der Aufgabenstellung für jeden Zeitpunkt übereinander an. Das sich ergebende Zustandsraster ist in Abbildung 5.1 für das Beispiel mit $P = (5, 8, 12)$ dargestellt.

Dieses Zustandsraster bietet sich nun an, einen entsprechenden Lösungsansatz zu kontruieren. Befindet sich Petra in einer Zelle (i,t), also zur Zeit t in Abschnitt i, kann sie entweder warten, bewegt sich also in die Zelle $(i,t+1)$ oder nach links beziehungsweise rechts laufen und sich damit in die Zellen $(i+1,t)$ oder $(i-1,t)$ ausbreiten. In diesen Regeln ist bereits enthalten, dass

Algorithmus 10 Brute-Force-Ansatz basierend auf Breitensuche

Require: Problem $P = (p_1, \ldots, p_n)$
  Initialisiere *Aktuelle Liste* $A \leftarrow []$ und *Merkliste* $M \leftarrow []$
  Zeit $t \leftarrow 0$
  while Zielabschnitt n nicht in A **do**
    if A leer **and** M leer **then** ▷ Kein Zustand war erreichbar
      Füge 1 in A ein und setze $t \leftarrow$ NÄCHSTEÖFFNUNG$(1,t)$
    else if A leer **then** ▷ Zum aktuellen Zeitpunkt ist alles abgearbeitet
      Tausche A und M
      $t \leftarrow t+1$
    else
      Nehme erste Position i aus A
      Füge $i+1$, $i-1$ in A wenn die Zellen zur Zeit t OFFEN und nicht besucht sind
      Füge i in M wenn Zelle zur Zeit $t+1$ offen ist
    end if
  end while

sich Petra nur vorwärts in der Zeit bewegen kann, also von einer Zelle (i,t) niemals eine Zelle (j,t') zur bereits verganenen Zeit $t' < t$ erreicht. Die jeweiligen Entscheidungen sind natürlich nur erlaubt, wenn die entsprechende Zelle den Status OFFEN hat. Dieser einfache Ansatz ist bereits geeignet, Instruktionen zum Finden des schnellsten Wegs zum Grabmal aufzustellen. Dafür müssen lediglich alle möglichen Schritte zu einem Zeitpunkt t ausgeführt werden, bevor die am folgenden Zeitpunkt $t+1$ probiert werden, wie in Algorithmus 10 durch Einfügen in die aktuelle Liste A für Aufgaben zum gleichen Zeitschritt oder Einfügen in die Merkliste M für Aufgaben zum nächsten Zeitschritt erreicht wird. Das gewonnene Verfahren erinnert an eine Breitensuche, wenn man die betretbaren Zellen des Zustandsrasters als Knoten ansieht und alle voneinander erreichbaren Knoten miteinander verbindet.

Die Schwäche des in Algorithmus 10 beschriebenen Verfahrens ist es, dass es zu jedem Zeitschritt alle Abschnitte behandelt. Basierend auf dem Ansatz stellen wir im Folgenden zwei asymptotisch äquivalente Verbesserungen vor, die sich jedoch durch ihre Sichtweise auf das Problem soweit unterscheiden, dass wir sie getrennt behandeln. Diese sind in Abbildung 5.2 für das Labyrinth in `grabmal1.txt` dargestellt.

Scanline-Ansatz

Im ersten Verfahren ist die Kernidee, sofort zu dem Zeitpunkt zu springen, bei welchem sich auf oder direkt neben einem besuchbaren Abschnitt der jeweilige Status ändert. In jedem Schritt wird dann lediglich mitgeführt, auf welchen Feldern Petra stehen könnte; dies ist etwa durch einen Array möglich. Der in Algorithmus 11 skizzierte beziehungsweise in Abbildung 5.2 illustrierte Scanline-Ansatz muss nun lediglich die Zeitpunkte und Abschnitte behandeln, an denen Änderungen stattfinden.

Um effizient herauszufinden, welche Zelle als nächstes geändert wird, benutzen wir eine Prioritätenwarteschlange, in welche der Zeitpunkt der Änderung sowie die sich ändernde Zelle vermerkt und entsprechend des Zeitpunkts priorisiert werden. Sobald eine Zelle als begehbar vermerkt wird, wird ihre nächste Änderung sowie alle Änderungen ihrer Nachbarn bis zu ihrem Schließen dort hinterlegt.

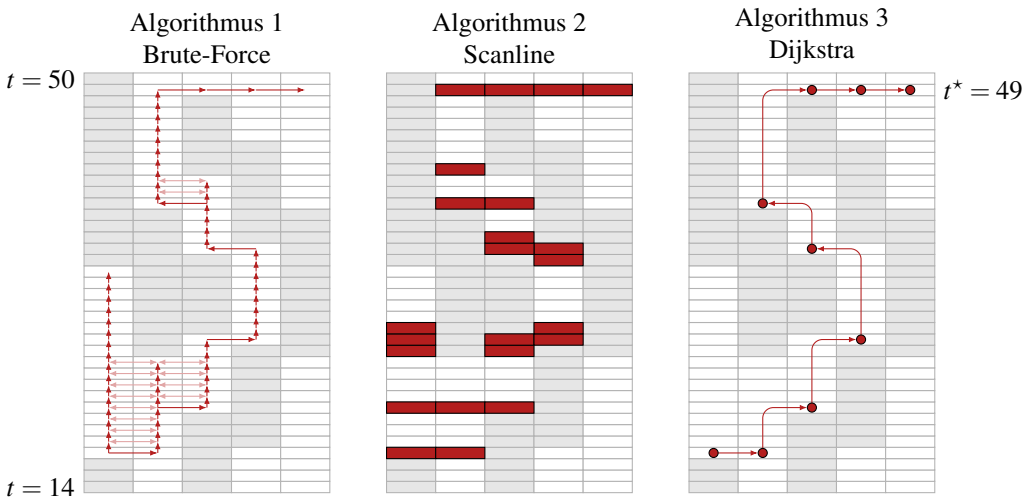


Abbildung 5.2: Vorgehen der Algorithmen 10 bis 12 anhand des kürzesten Wegs in `grabmal1.txt`, dargestellt von $t = 14$ bis $t = 50$; bei $t^\star = 49$ kann Petra das Grabmal erreichen. Für den Brute-Force-Ansatz sind alle betrachteten Bewegungen (rote Pfeile) entsprechend der festgelegten Sprungregeln von Zelle (i,t) in die Zellen $(i,t+1)$ oder $(i\pm1,t)$ vermerkt. Hier werden alle betretbaren Felder besucht, wohingegen Scanline- und Dijkstra-Ansatz nur zu den relevanten Zeitpunkten und Zellen springen. Für Algorithmus 11 bedeutet das, jeden Zeitpunkt, bei dem ein Abschnitt mit einer erreichbaren Zelle (rot markiert) oder einer seiner Nachbarn den Zustand ändert. Für Algorithmus 12 sind die berechneten Knoten (rote Punkte) und Kanten (rote Pfeile) eingezeichnet.

Dijkstra-Ansatz

Anstatt zu jedem Zeitpunkt mitzuschreiben, welche Zellen gerade betretbar sind, können die Zellen, bei denen ein Abschnitt das erste Mal betreten werden kann, als Knoten in einem Graphen aufgefasst werden. Diese Knoten werden miteinander durch eine Kante verbunden, sobald eine jeweilige Zelle durch Petra von der anderen aus betreten werden kann. Das Ziel, schnellstmöglich das Grabmal zu erreichen, kann dann mit einer Abwandlung des Algorithmus von Dijkstra gelöst werden, bei dem die aktuelle Distanz durch die aktuell vergangene Zeit gegeben ist. Hier gilt es allerdings zu beachten, dass der vollständige Graph, auf dem die Suche stattfindet, nie vorliegt; dieser wäre schlichtweg viel zu groß. Stattdessen werden beim Betreten eines Knotens dessen Nachbarknoten berechnet und vermerkt, die sich durch Bewegen nach links oder rechts ergeben. Wird ein Zielknoten, also eine beliebige Zelle im letzten Abschnitt, erreicht, ist die Wegsuche abschlossen und der kürzeste Weg zum Grabmal gefunden.

Das Verfahren ist in Algorithmus 12 als Pseudocode und wieder in Abbildung 5.2 als Illustration gegeben. Die Liste Q in Algorithmus 12 ist dabei wie zuvor eine Prioritätenwarteschlange.

Aufstellen der Instruktionen

Die weiter oben beschriebenen Verfahren geben als solche noch keine Instruktionen zum Finden des kürzesten Wegs aus. Um dies zu ermöglichen, lassen sich alle Algorithmen durch das Mitführen einer Liste erweitern, welche die Weghistorie mitschreibt. Für jeden Abschnitt beziehungsweise Knoten im Fall von Algorithmus 12 wird dabei vermerkt, wie Petra zum aktuellen

Algorithmus 11 Scanline Ansatz

Require: Problem $P = (p_1, \ldots, p_n)$
  Vermerke alle Abschnitte als nicht betretbar
  Zeit $t \leftarrow 0$
  while Zielabschnitt n nicht betretbar **do**
    Gehe zu Zeit $t' \geq t$, bei der ein betretbarer Abschnitt, ein von einem betretbaren benachbarter Abschnitt, oder der erste Abschnitt seinen Status ändern
    if geänderter Abschnitt ist jetzt OFFEN **then**
      Vermerke den Abschnitt und alle neu erreichbaren als betretbar
    else
      Vermerke den Abschnitt als nicht betretbar
    end if
  end while

Algorithmus 12 Dijkstra Ansatz

Require: Problem $P = (p_1, \ldots, p_n)$
  Initialisiere Liste $Q \leftarrow []$
  while Zielabschnitt n nicht in Q **do**
    if Q leer **or** Öffnung von Abschnitt 1 übersprungen **then**
      Nehme Paar $(1,t)$, wobei t die nächste anstehende Öffnung von Abschnitt 1 ist
    else
      Nehme Paar (i,t) aus Q, wobei t die kleinste nächste Zeit ist
    end if
    Füge alle links und rechts bis zur Schließung von i erreichbaren, unbesuchten Abschnitte und Zeiten in Q ein
  end while

Zustand gekommen ist. In Algorithmus 10 kann man beispielsweise aufzeichnen, welche Zellen vorher besucht wurden. Aus diesen Informationen können dann einfach die Instruktionen generiert werden, in dem bei einer Zeitdifferenz „Warte“ und einer Ortsdifferenz „Gehe zu“ ausgegeben werden.

Laufzeitanalyse

Im Worst-Case besucht Algorithmus 10 zu jeder möglichen Zeit jeden Abschnitt, hat also eine Laufzeit von $\mathcal{O}(nT(P))$. Sowohl Algorithmus 11 als auch Algorithmus 12 verbessern diese asymptotische Laufzeit, in dem sie für jeden Abschnitt nur die Zeitpunkte betrachten, in denen dieser seinen Status ändert, also $\mathcal{O}(\sum_i T(P)/2p_i)$. In der Praxis sind die Grabmale allerdings sehr viel eher als zur Zeit $T(P)$ erreichbar. Einfügen und Entfernen des jeweils nächsten geänderten Abschnitts in Algorithmus 11 beziehungsweise nächsten zu besuchenden Knotens in Algorithmus 12 in der Prioritätenwarteschlange benötigt die Laufzeit $\mathcal{O}(\log n)$.

5.2 Erweiterungen

Im bisherigen Teil wurde lediglich die benötigte Zeit ins Ziel als Spezifikation für die Definition der kürzesten Route angesehen. Allerdings können noch weitere Anforderungen an die

Instruktionen gestellt werden, um diese Petrafreundlicher zu gestalten. Hierbei wird es sehr individuell, dementsprechend beschreiben wir die von den Algorithmen 11 und 12 erzeugten Instruktionen und skizzieren mögliche andere Anforderungen. Die Algorithmen 11 und 12 zeichnen sich dadurch aus, dass sie jeden Abschnitt so schnell wie möglich auf dem Weg zum Grabmal wieder verlassen. Dadurch wird die Anforderung erfüllt, dass Petra so sich so kurz wie möglich gedulden muss und ihr nächster Schritt immer der erste mögliche auf dem schnellsten Weg zum Grabmal ist. Alternativ könnte man auch die Anforderung stellen, dass Petra so wenig Instruktionen wie möglich erhält. Hier müsste man den Algorithmus so anpassen, dass für zwei aufeinanderfolgende Anweisungen mit jeweils einer Wartezeit geprüft wird, ob Petra auch einfach eine längere Wartezeit absitzen kann und dann immer noch auf dem Weg das Grabmal erreichen kann. Hierdurch könnten sukzessive Instruktionen vereinigt werden, wodurch sich ihre Anzahl reduziert.
Zusätzlich kann beispielsweise die Beweglichkeit von Petra eingeschränkt werden; so können wir beispielsweise einfügen, dass Petra in einer Minute nur einen Schritt nach links oder rechts vornehmen kann. Dadurch entfällt die oben bewiesene Lösbarkeit, und die Ausbreitungsschritte des Algorithmus müssen um eine Erhöhung der Zeit beim Besuchen der Nachbarn erweitert werden. Der Algorithmus kann allerdings weiterhin eine schnellste Lösung finden, insofern diese existiert, und auch beweisen, ob ein Rätsel lösbar oder unlösbar ist. Für die Lösbarkeit kann immer nach $T(P)$ Zeitschritten vermerkt werden, auf welchen Abschnitten Petra stehen kann. Taucht eine bereits gesehene Konfiguration von Abschnitten erneut auf, wäre der Algorithmus in einer endlosen Schleife gefangen aus und das Grabmal wäre für Petra nicht erreichbar. Allerdings existieren $\mathcal{O}(2^n)$ solcher Konfigurationen, wodurch diese Beweisstrategie für große Rätsel unpraktikabel wird.
Ebenfalls möglich wäre die Erweiterung, dass das Labyrinth kein Gang mehr ist, sondern ein Graph dessen Knoten die jeweiligen Abschnitte sind. Auch hier können nach einer kleinen Modifikation die obigen Algorithmen verwendet werden. Anstatt sich beim Bewegen auf den linken und rechten Abschnitt zu beschränken, können nur die im Graphen spezifizierten Nachbarn angesteuert werden.

5.3 Beispiele

In `grabmal0.txt` kann das Grabmal nach **12 Minuten** erreicht werden.
Die entsprechenden Anweisungen sind: Warte 5 Minuten, laufe zum Abschnitt 1, warte 3 Minuten, laufe zum Abschnitt 2, warte 4 Minuten, laufe zum Grabmal.

In `grabmal1.txt` kann das Grabmal nach **49 Minuten** erreicht werden.
Die entsprechenden Anweisungen sind: Warte 17 Minuten, laufe zum Abschnitt 1, laufe zum Abschnitt 2, warte 4 Minuten, laufe zum Abschnitt 3, warte 6 Minuten, laufe zum Abschnitt 4, warte 8 Minuten, laufe zum Abschnitt 3, warte 4 Minuten, laufe zum Abschnitt 2, warte 10 Minuten, laufe zum Abschnitt 3, laufe zum Abschnitt 4, laufe zum Grabmal.

In `grabmal2.txt` kann das Grabmal nach **490 021 Minuten** erreicht werden.
Die entsprechenden Anweisungen sind: Warte 170 000 Minuten, laufe zum Abschnitt 1, laufe zum Abschnitt 2, warte 40 009 Minuten, laufe zum Abschnitt 3, warte 59 991 Minuten, laufe zum Abschnitt 4, warte 80 015 Minuten, laufe zum Abschnitt 3, warte 39 985 Minuten, laufe zum Abschnitt 2, warte 100 021 Minuten, laufe zum Abschnitt 3, laufe zum Abschnitt 4, laufe zum Grabmal.

In `grabmal3.txt` kann das Grabmal nach **35 Minuten** erreicht werden.
Die entsprechenden Anweisungen sind: Warte 20 Minuten, laufe zum Abschnitt 1, warte 2

Minuten, laufe zum Abschnitt 2, laufe zum Abschnitt 3, laufe zum Abschnitt 4, laufe zum Abschnitt 5, laufe zum Abschnitt 6, warte 13 Minuten, laufe zum Abschnitt 7, laufe zum Abschnitt 8, laufe zum Abschnitt 9, laufe zum Grabmal.

In `grabmal4.txt` kann das Grabmal nach **3 432 897 Minuten** erreicht werden.
Die entsprechenden Anweisungen sind: Warte 3 242 835 Minuten, laufe zum Abschnitt 1, warte 60 360 Minuten, laufe zum Abschnitt 2, warte 360 Minuten, laufe zum Abschnitt 3, laufe zum Abschnitt 4, laufe zum Abschnitt 5, warte 37 695 Minuten, laufe zum Abschnitt 6, warte 51 407 Minuten, laufe zum Abschnitt 7, laufe zum Abschnitt 8, warte 36 216 Minuten, laufe zum Abschnitt 9, warte 4024 Minuten, laufe zum Grabmal.

In `grabmal5.txt` kann das Grabmal nach **43 965 990 Minuten** erreicht werden.
Die entsprechenden Anweisungen sind: Warte 43 838 148 Minuten, laufe zum Abschnitt 1, warte 45 Minuten, laufe zum Abschnitt 2, laufe zum Abschnitt 3, laufe zum Abschnitt 4, laufe zum Abschnitt 5, warte 4542 Minuten, laufe zum Abschnitt 6, warte 6752 Minuten, laufe zum Abschnitt 7, warte 1452 Minuten, laufe zum Abschnitt 8, warte 3576 Minuten, laufe zum Abschnitt 9, warte 281 Minuten, laufe zum Abschnitt 10, laufe zum Abschnitt 11, laufe zum Abschnitt 12, laufe zum Abschnitt 13, laufe zum Abschnitt 14, warte 5969 Minuten, laufe zum Abschnitt 15, laufe zum Abschnitt 16, laufe zum Abschnitt 17, warte 2246 Minuten, laufe zum Abschnitt 18, warte 2629 Minuten, laufe zum Abschnitt 19, warte 1120 Minuten, laufe zum Abschnitt 20, laufe zum Abschnitt 21, warte 2408 Minuten, laufe zum Abschnitt 22, warte 5247 Minuten, laufe zum Abschnitt 23, warte 1620 Minuten, laufe zum Abschnitt 24, warte 825 Minuten, laufe zum Abschnitt 25, warte 3438 Minuten, laufe zum Abschnitt 26, laufe zum Abschnitt 27, warte 1034 Minuten, laufe zum Abschnitt 28, warte 349 Minuten, laufe zum Abschnitt 29, laufe zum Abschnitt 30, laufe zum Abschnitt 31, laufe zum Abschnitt 32, warte 1694 Minuten, laufe zum Abschnitt 33, laufe zum Abschnitt 34, warte 2894 Minuten, laufe zum Abschnitt 35, warte 4867 Minuten, laufe zum Abschnitt 36, warte 70 Minuten, laufe zum Abschnitt 37, laufe zum Abschnitt 38, laufe zum Abschnitt 39, warte 1706 Minuten, laufe zum Abschnitt 40, laufe zum Abschnitt 41, laufe zum Abschnitt 42, laufe zum Abschnitt 43, warte 204 Minuten, laufe zum Abschnitt 44, laufe zum Abschnitt 45, warte 859 Minuten, laufe zum Abschnitt 46, warte 195 Minuten, laufe zum Abschnitt 47, warte 2040 Minuten, laufe zum Abschnitt 48, warte 1751 Minuten, laufe zum Abschnitt 49, warte 4079 Minuten, laufe zum Abschnitt 50, warte 1257 Minuten, laufe zum Abschnitt 51, warte 731 Minuten, laufe zum Abschnitt 52, warte 1001 Minuten, laufe zum Abschnitt 53, warte 4342 Minuten, laufe zum Abschnitt 54, warte 1332 Minuten, laufe zum Abschnitt 55, warte 4533 Minuten, laufe zum Abschnitt 56, warte 3474 Minuten, laufe zum Abschnitt 57, warte 113 Minuten, laufe zum Abschnitt 58, laufe zum Abschnitt 59, warte 2935 Minuten, laufe zum Abschnitt 60, laufe zum Abschnitt 61, warte 1112 Minuten, laufe zum Abschnitt 62, warte 289 Minuten, laufe zum Abschnitt 63, warte 3931 Minuten, laufe zum Abschnitt 64, warte 704 Minuten, laufe zum Abschnitt 65, laufe zum Abschnitt 66, laufe zum Abschnitt 67, laufe zum Abschnitt 68, warte 481 Minuten, laufe zum Abschnitt 69, warte 4319 Minuten, laufe zum Abschnitt 70, laufe zum Abschnitt 71, warte 5643 Minuten, laufe zum Abschnitt 72, warte 3633 Minuten, laufe zum Abschnitt 73, warte 413 Minuten, laufe zum Abschnitt 74, warte 4375 Minuten, laufe zum Abschnitt 73, warte 617 Minuten, laufe zum Abschnitt 72, warte 1906 Minuten, laufe zum Abschnitt 71, warte 7132 Minuten, laufe zum Abschnitt 72, warte 3113 Minuten, laufe zum Abschnitt 73, warte 651 Minuten, laufe zum Abschnitt 74, warte 792 Minuten, laufe zum Abschnitt 75, laufe zum Abschnitt 76, laufe zum Abschnitt 77, laufe zum Abschnitt 78, laufe zum Abschnitt 79, laufe zum Abschnitt 80, warte 2414 Minuten, laufe zum Abschnitt 81, warte 1222 Minuten, laufe zum Abschnitt 82, warte 1485 Minuten, laufe zum Abschnitt 83, laufe zum Abschnitt 84, laufe zum Grabmal.

5.4 Bewertungskriterien

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

- $[-1]$ **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**

- $[-1]$ **Modellierung ungeeignet**
  Sowohl Petras Situation als auch der Zustand des Labyrinths bzw. der Quader im Gang zum Grabmal müssen sinnvoll modelliert werden. Die vergangene Zeit und die jeweiligen Perioden der Steine genügen für die Darstellung des Labyrinths. Es muss möglich sein, zwischen zugänglichen und nicht-zugänglichen Abschnitten zu unterscheiden.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das Verfahren muss berücksichtigen, dass Petra einen Abschnitt zurückgehen kann. Die zum Erreichen des Grabmals benötigte Zeit muss jeweils der schnellstmöglichen Zeit entsprechen. . Die Abfolge von Instruktionen muss möglich sein.

  Es sollte Petra nicht möglich sein, durch verschlossene Abschnitte zu laufen oder sich in diesen aufzuhalten. Petra darf außerdem nicht durch die Zeit reisen.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Die Anweisungen für Petra sollten in höchstens $\mathcal{O}(nt)$ Schritten aufgestellt werden, wobei n die Anzahl der Abschnitte und t die schlussendliche Zeit zum Grabmal ist.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Die Ergebnisse müssen die Instruktionen zum Grabmal enthalten. Die insgesamt benötigte Zeit muss mit Hilfe der Ausgabe erschließbar sein. Für eine bessere Lesbarkeit dürfen *Laufe zu* Anweisungen ohne Wartezeit dazwischen zusammengefasst werden.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu allen sechs vorgegebenen Beispielen (`grabmal0.txt` bis `grabmal5.txt`) enthalten.

Beispiel	Benötigte Minuten
0	12
1	49
2	490.021
3	35
4	3.432.897
5	43.965.990

Exkurs zur Junioraufgabe 1: Quadratisch, praktisch, grün

In diesem Abschnitt werden Verbesserungen des Lösungsansatzes beschrieben, die nicht notwendig sind, um die Aufgabe vollständig zu lösen. Solche fortgeschrittenen Ansätze sind jedoch oft in der zweiten Runde des Bundeswettbewerbs Informatik nützlich oder notwendig, um die Aufgaben lösen zu können.

Den Lösungsansatz aus 1 können wir verbessern, in dem wir umstellen:

$$\begin{aligned} & k \leq nm \leq 1.1k && \mid : n \\ \leftrightarrow \quad & \frac{k}{n} \leq m \leq \frac{1.1k}{n} && \mid \text{Weil } m \in \mathbb{N}^+ \\ \leftrightarrow \quad & \left\lceil \frac{k}{n} \right\rceil \leq m \leq \left\lfloor \frac{1.1k}{n} \right\rfloor \end{aligned}$$

Somit müssen wir für ein n nicht über alle $1 \leq m \leq 1.1k$ iterieren, sondern nur über die, welche die Ungleichung erfüllen. Diese neuen Intervallgrenzen können wir in der Brute-Force-Prozedur an der Stelle einsetzen, an der über alle möglichen m iteriert wird.

Wir können es uns sogar sparen, für ein n über alle erlaubten m zu iterieren, um das Optimum zu finden. Im folgenden Abschnitt sei also n fest, und wir suchen für dieses feste n das erlaubte m, welches die quadratischsten Gärten erzeugt. Wenn H und B die Höhe und Breite des

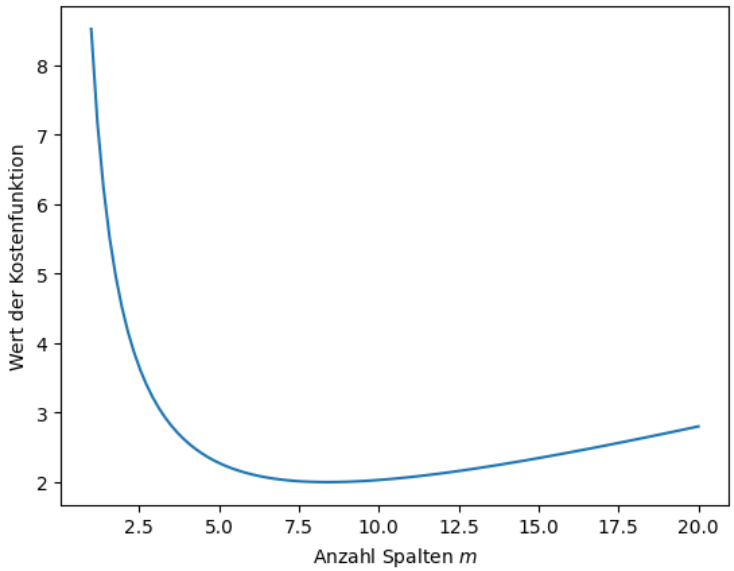


Abbildung 5.1: Kosten $f(\frac{H}{n}, \frac{B}{m})$ in Abhängigkeit von m für feste Höhe H, Breite B und Zeilenzahl n. Wie man sieht, gibt es nur ein globales Minimum.

Grundstückes sind, dann sind $\frac{H}{n}$ und $\frac{B}{m}$ die Höhe und Breite eines einzelnen Gartens. Wir wissen, dass die Kostenfunktion für eine feste Zeilenzahl (also Höhe) genau dann mimimiert wird, wenn die Gärten genau so breit wie hoch sind. Damit können wir die optimale Spaltenzahl m

berechnen:

$$\begin{aligned} & \frac{H}{n} = \frac{B}{m} && |\text{Kehrwert} \\ \leftrightarrow \quad & \frac{n}{H} = \frac{m}{B} && |\cdot B \\ \leftrightarrow \quad & \frac{nB}{H} = m \end{aligned}$$

In Abbildung 5.1 sehen wir, dass dieser Tiefpunkt für bestimmte H, B, n der einzige ist, und dass die Kosten überall vor dem Tiefpunkt fallend und überall danach steigend sind. Um zu prüfen, ob das für alle H, B, n gilt, müssen wir die Ableitung von $f(\frac{H}{n}, \frac{B}{m}) = \frac{Hm}{Bn} + \frac{Bn}{Hm} = \frac{H}{Bn}m + \frac{Bn}{H}m^{-1}$ über m bilden, denn diese gibt die Steigung der Kosten für jeden Punkt an. Durch Anwendung der Summen-, Potenz-, und Faktorregel für Ableitungen[18] erhalten wir die Ableitung $\frac{H}{Bn} - \frac{Bn}{H}m^{-2}$. Wenn wir den berechneten Tiefpunkt hier einsetzen, erhalten wir erwartungsgemäß 0. Anhand dieser Ableitung können wir nun prüfen, wann die Funktion steigend (Ableitung größer als 0) oder fallend (Ableitung kleiner als 0) ist. Wir setzen also die Ableitung größer als 0:

$$\begin{aligned} & 0 < \frac{H}{Bn} - \frac{Bn}{H}m^{-2} && |+\frac{Bn}{H}m^{-2} \\ \leftrightarrow \quad & \frac{Bn}{H}m^{-2} < \frac{H}{Bn} && |\text{Kehrwert (Ungleichung wird umgekehrt)} \\ \leftrightarrow \quad & \frac{H}{Bn}m^2 > \frac{Bn}{H} && |:\frac{H}{Bn} \\ \leftrightarrow \quad & m^2 > \frac{(Bn)^2}{H^2} && |\text{Quadratwurzel} \\ \leftrightarrow \quad & m > \frac{Bn}{H} \end{aligned}$$

Die Kosten steigen also genau dann, wenn m größer als der Tiefpunkt ist. Analog dazu können wir durch Vertauschen der Ungleichungszeichen ermitteln, dass die Kosten genau dann sinken, wenn m kleiner als der Tiefpunkt ist. Somit gibt es auch nur einen Tiefpunkt.

Die Intervallgrenzen für erlaubte m ($\lceil \frac{k}{n} \rceil \leq m \leq \lfloor \frac{1.1k}{n} \rfloor$) liegen dadurch entweder beide auf der gleichen Seite des globalen Tiefpunktes oder auf beiden Seiten, so dass der globale Tiefpunkt enthalten ist. Falls beide auf der gleichen Seite des Tiefpunktes liegen, nehmen wir einfach die Grenze mit kleineren Kosten als bestes m, denn es gibt in dem Intervall keinen lokalen Tiefpunkt, und wenn die Grenzen den Tiefpunkt einschließen, nehmen wir diesen als bestes m. Wenn der Tiefpunkt keine natürliche Zahl ist, runden wir ihn auf und ab und vergleichen die Kosten dieser beiden Spaltenzahlen. Durch das Steigungsverhalten ist das Bessere von den beiden optimal. Wir haben also entweder die obere und untere Intervallgrenze als Kandidaten für die Spaltenzahl oder den auf- und abgerundeten Tiefpunkt. Dieser Ansatz ergibt die in Algorithmus 13 notierte Prozedur zur Berechnung der Spaltenzahl. Diese Prozedur können wir auch in das Brute-Force-Verfahren von oben an der Stelle einsetzen, wo über alle möglichen m iteriert wird.

Wir können noch mehr Rechenschritte mit einem Verfahren sparen, das nur Aufteilungen findet, in denen die Gärten nicht breiter als hoch sind. Dieses Verfahren können wir zuerst auf das ursprüngliche Grundstück anwenden, und dann auf ein neues Grundstück, bei dem einfach Hö-

[18] `https://de.wikibooks.org/wiki/Mathe_f%C3%BCr_Nicht-Freaks:_Ableitungsregeln:_Kettenregel,_Quotientenregel,_Produktregel,_Summenregel,_Faktorregel`

Algorithmus 13 Optimale Spaltenanzahl

```
procedure OPTIMALE SPALTENZAHL(Höhe H, Breite B, Interessenten k, Zeilen n)
    if ⌈k/n⌉ > ⌊1.1 k/n⌋ then return nichts
    end if
    if k/n ≤ nB/H ≤ 1.1 k/n then
        m1 ← ⌊nB/H⌋
        m2 ← ⌈nB/H⌉        ▷ m1 und m2 sind die beiden Kandidaten für die optimale Spaltenzahl
        if m1 < k/n then
            m1 ← m2
        end if
        if m2 > 1.1 k/n then
            m2 ← m1
        end if
    else
        m1 ← ⌈k/n⌉
        m2 ← ⌊1.1 k/n⌋
    end if
    if f(H/n, B/m1) < f(H/n, B/m2) then
        return m1
    else
        return m2
    end if
end procedure
```

he und Breite vom ursprünglichen getauscht sind, wodurch wir trotz der Einschränkung keine Aufteilungen des Grundstückes aussschließen. Es muss also $\frac{H}{n} \geq \frac{B}{m}$ gelten. Wir formen um:

$$
\begin{aligned}
& \frac{H}{n} \geq \frac{B}{m} && | \cdot m \\
\leftrightarrow \quad & \frac{H}{n} m \geq B && | : \frac{H}{n} \\
\leftrightarrow \quad & m \geq \frac{nB}{H} && | \text{Weil } \frac{1.1k}{n} \geq m \text{ und } \geq \text{ transitiv ist} \\
\rightarrow \quad & \frac{1.1k}{n} \geq \frac{nB}{H} && | \cdot \frac{nH}{B} \\
\leftrightarrow \quad & \frac{1.1kH}{B} \geq n^2 && | \text{Quadtratwurzel} \\
\leftrightarrow \quad & \sqrt{\frac{1.1kH}{B}} \geq n && | \text{Weil } n \in \mathbb{N} \\
\leftrightarrow \quad & \left\lfloor \sqrt{1.1 \frac{kH}{B}} \right\rfloor \geq n
\end{aligned}
$$

Das ist eine neue Obergrenze für n, weshalb wir für n nicht alle Werte bis $\lfloor 1.1k \rfloor$ sondern nur bis zu dieser neuen oberen Grenze probieren müssen, falls sie niedriger ist. Für ein bestimmtes n können wir dann die Prozedur zur Berechnung von m von oben benutzen. Diese müssen wir nicht modifizieren, um Gärten, die breiter als hoch sind, auszuschließen, denn diese sind

immernoch gültige Lösungen für unser ursprüngliches Problem.

Laufzeit der Verbesserungen

Mit der ersten Verbesserung probieren wir für jeden n-Wert alle erlaubten m-Werte durch, das heißt wir probieren insgesamt

$$\begin{aligned}\sum_{n=1}^{\lfloor 1.1k\rfloor}\left(\left\lfloor\frac{1.1k}{n}\right\rfloor-\left\lceil\frac{k}{n}\right\rceil\right) &\leq \sum_{n=1}^{\lfloor 1.1k\rfloor}\left(\frac{1.1k}{n}-\frac{k}{n}\right)\\ &= 0.1k\sum_{n=1}^{\lfloor 1.1k\rfloor}\frac{1}{n} && \mid \text{Harmonische Folge}\\ &= \mathcal{O}(k\log k)\end{aligned}$$

Gitter durch. Das Verfahren braucht also die Zeit $\mathcal{O}(k\log k)$

Nach der zweiten Verbesserung prüfen wir für jeden n-Wert nur kostant viele m-Werte. Da wir $\lfloor 1.1k\rfloor$ verschiedene n-Werte probieren, läuft dieses Verfahren in der Zeit $\mathcal{O}(k)$.

Im letzten Ansatz probieren wir $\min(\lfloor 1.1k\rfloor,\lfloor\sqrt{1.1\frac{kH}{B}}\rfloor)+\min(\lfloor 1.1k\rfloor,\lfloor\sqrt{1.1\frac{kB}{H}}\rfloor)$ verschiedene Werte für n und haben jeweils konstanten Aufwand für jeden n-Wert, wodurch wir eine Laufzeit von

$$\begin{aligned}&\mathcal{O}(\min(\lfloor 1.1k\rfloor,\lfloor\sqrt{1.1\frac{kH}{B}}\rfloor)+\min(\lfloor 1.1k\rfloor,\lfloor\sqrt{1.1\frac{kB}{H}}\rfloor))\\ &=\mathcal{O}(\min(\lfloor 1.1k\rfloor,\lfloor\sqrt{1.1\frac{kH}{B}}\rfloor+\lfloor\sqrt{1.1\frac{kB}{H}}\rfloor))\\ &=\mathcal{O}(\min(1.1k,\sqrt{1.1}\sqrt{k}\sqrt{\frac{H}{B}+\frac{B}{H}}))\end{aligned}$$

haben.

Abschließend sei erwähnt, dass man sich in der Informatik normalerweise auch für die Laufzeit in Abhängigkeit von der Eingabelänge interessieren. Da wir nur ca. $\log_2 k$ Bits brauchen, um die Zahl k dazustellen, wächst k also exponentiell zur Eingabelänge.Wenn wir nun $k=2^n$, $H=2^m$ und $B=2^l$ in die Laufzeiten oben einsetzen, erhalten wir für den Bruteforce-Ansatz $\mathcal{O}(k^2)=\mathcal{O}((2^n)^2)=\mathcal{O}((2^{2n}))$, für die erste Verbesserung $\mathcal{O}(k\log k)=\mathcal{O}(2^n\log 2^n)=\mathcal{O}(n2^n)$, für die zweite Verbesserung $\mathcal{O}(k)=\mathcal{O}(2^n)$ und für die letzte Verbesserung

$$\begin{aligned}\mathcal{O}(\min(1.1k,\sqrt{1.1}\sqrt{k}\sqrt{\frac{H}{B}+\frac{B}{H}})) &= \mathcal{O}(\min(1.1\cdot 2^n,\sqrt{1.1}\sqrt{2^n}\sqrt{\frac{2^m}{2^l}+\frac{2^l}{2^m}}))\\ &= \mathcal{O}(\min(1.1\cdot 2^n,\sqrt{1.1}\cdot 2^{\frac{n}{2}}\sqrt{2^{m-l}+2^{l-m}}))\end{aligned}$$

Die vorgestellten Algorithmen laufen also alle in exponentieller Zeit zur Eingabelänge. Wenn ein Algorithmus zwar zur eingegebenen Zahl polynomielle Zeit, aber zur Eingabelänge exponentielle Zeit braucht, spricht man von einem pseudopolynomiellen Algorithmus.

Aus den Einsendungen: Perlen der Informatik

Junioraufgabe 1: Quadratisch, praktisch, grün

[...] der Teiler der Menschen [...]

Es gibt einen Garten mit gegebenen Seitenlängen, der in einzelne Kleingärten aufgeteilt werden soll, wobei alle Kleingärten... Moment mal! An irgendetwas erinnert mich der Titel doch... Genau! An Ritter Sport-Schokolade. Deren Werbespruch ist doch 'Quadratisch, praktisch, gut', nicht grün... egal, weiter gehts mit der Lösungsidee. [...] So hat man dann ganz viele (fast) quadratische Schokostü...ääh...Kleingärten. [...] Vor dem Programmieren holt man sich am besten noch einen Snack, beispielsweise Schokolade, dann kann man sich besser konzentrieren und das Programmieren macht umso mehr Spaß:

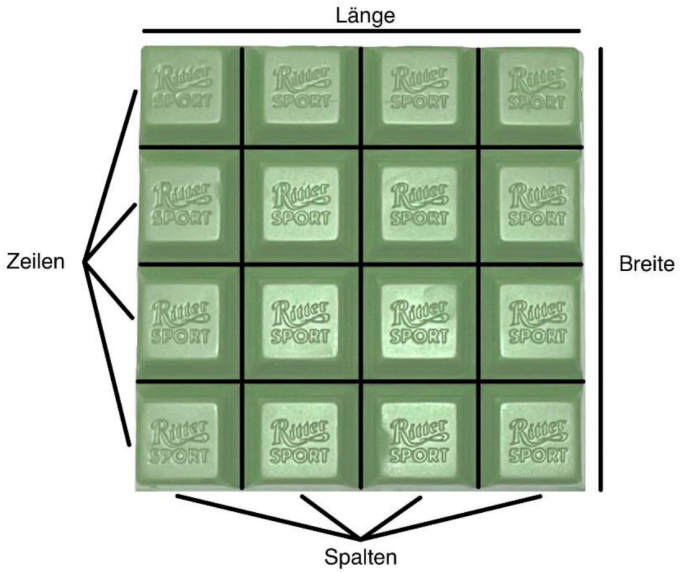


*Bild1: Quadratisch, praktisch, grün **und** gut: Eine Darstellung eines „Gartens" mit einer Einteilung in 4 Zeilen und 4 Spalten, also in 16 (leckere) Quadrate*
~ eigene Quelle

Die Interessenten in meinem Programm heißen Rentner, da ich früher mal Mieter als Rentner bezeichnet habe, da das englische Wort für Miete „rent" ist und ich mich vertan habe. Heute ist es ein Insider-Joke bei mir zuhause.

Danach wird für jede dieser Aufteilungen erst die Höhe durch Faktor 1 [...] geteilt.

Junioraufgabe 2: Texthopsen

```
print(„Es ist wichtig anzumerken: In dieser Situation wäre eigentlich Gleichstand,
weil beide gleich viele Sprünge gebraucht hätten. Fazit: die Freundschaft
hätte gewinnen sollen!!")
```

Im dritten Text war eine Zahl enthalten, aber ich war nach `hopsen2.txt` nicht mehr überrascht, dass etwas nicht funktionierte. Zahlen war wie Sonderzeichen, wer weiß was es noch für Ausnahmen gibt.

Eigene Beispiele:

„Edamer schmeckt gut!“, flüsterte Amira zu Bela. „Mag ich auch! Magst du Brötchen?“, fragte Bela. „Über was redet ihr?“, kam die Lehrerin, „Konzentriert euch auf die Aufgabenstellung! Ich werd euch gleich als Erstes dran nehmen, vorzustellen!“ „###:::f*“, murmelte Bela.

Aufgabe 1: Hopsitexte

Bei jeder neuen Texteingabe wird die Hopslogik in Echtzeit analysiert.

Mein Algorithmus ist recht simpel: Es wird eine Zahl n eingegeben, und dann mit einer for-Schleife entsprechend oft der Satz „Oo, wie lang das ist:“ ausgegeben. Der Vorteil hierbei ist, dass der Algorithmus sehr schnell ist.

Da ich nicht wusste, was genau „sinfoll deutsch“ ist [...]

Hier ein ca 230.000 Buchstaben langer Hopsitext in Form einer Beschreibung von Lars Gedanken (längere Texte lässt mein LaTeX-Anbieter nicht zu):
`Aal Lars dachte sich:  Aal Lars dachte sich:`
[Der Satzteil wird auf den nächsten 53 Seiten wiederholt und ist von Beginn an kein Hopsitext.]

Aaliger Quellcode:

```
n = int(input('Wie lang soll der Text mindestens sein?'))
for i in range(n):
print('Aal Lars dachte sich:  ')
```

Andere sinnvolle deutsche Texte:

- Das Flugzeug diskutiert schüchtern seit 53 Dekaden. Das Regal verkauft optimistisch seit 200 Jahrzehnten. Die Socke singt normal seit 87 Ewigkeiten. Die Kamera zieht teuer seit 143 Monatsenden. Der Fisch fängt schwer seit 89 Stundenbruchteilen. Das Einhorn begleitet müde seit 41 Halbtagen.
- Star Wars ist ein sehr populärer Film, der die Geschichte von Imperium Galactica erzählt.Er wurde von der Filmstiftung Nordrhein-Westfalen gefördert.Die Serie spielt im Jahr 2027 und ist in der Zeit zwischen dem Beginn der Handlung und dem Ende der Geschichte angesiedelt.Die Geschichte beginnt mit einem Angriff der imperialen Flotte auf eine geheime Stadt.Der Spieler steuert die Figur des Imperators Galatea, die aus dem Spiel heraus die Welt beherrscht und die Erde in ihrer Existenz bedroht.Diese Bedrohung wird durch den Angriff des letzten Imperiums abgewehrt.Es folgt eine Reihe von Missionen, in denen der Spieler die Rolle des Protagonisten übernimmt.Dabei übernimmt er verschiedene Rollen, wie zum Beispiel die des Bösewichts.In dieser Zeit wird er von einem Team des Militärs gejagt, welches die Stadt von seinem Planeten kontrolliert.Nach einem weiteren Angriff auf die Imperiale Stadt, wird der Imperator von dem Team von Admiral Zod getötet.Dieser versucht nun, das Team zu besiegen, was jedoch misslingt.Schließlich kommt es zum Kampf zwischen Zods Crew und den anderen Spielern. Zodo, ein Mitglied des Teams, versucht, den Feind zu töten, scheitert jedoch.Danach übernimmt das Imperio-Team die Kontrolle über die Flotte und besiegt den Gegner.Das Imperiosystem wird von den Spielern des Spiels übernommen.Im Spielverlauf wird das Spiel durch die Verwendung von verschiedenen Waffen und Waffenkombinationen verbessert.So kann der Soldat z.B.die Handgranate mit der Hand des Gegners in den Händen

halten.Außerdem kann er sich in das feindliche Schiff einwählen, um sich zu verteidigen.Auch kann man mit dem Imperius-Schiff in die Galaxis zurückkehren.Ein weiteres Spielelement ist das sogenannte SScooby-Doo-Spiel".Dieses ist eine Art Spiel, bei dem man verschiedene Gegenstände mit den gleichen Zielen und Zielen benutzen kann.

Aufgabe 2: Schwierigkeiten

Wir haben im ersten Schritt unsere Aufgabe und dessen Aufgabenstellung angesehen und versucht sie so weit wie möglich zu verstehen Mithilfe kleiner Karteikärtchen mit den Buchstaben von A bis N, konnten wir uns bildlich die Aufgabe auflegen lassen und die Karten so verschieben, dass es am Ende passt.

Meine Lösung basiert auf einem Sortieralgorithmus, da ich die Aufgaben nach Schwierigkeit anordnen muss. Es gibt viele verschiedene Sortierverfahren, und theoretisch hätte ich sogar ein eigenes entwickeln können - aber warum das Rad neu erfinden?

Sortieren aller Aufgaben durch Physiksimulation - Als Überschrift eines Abschnitts.

Aufgabe 3: Wandertag

Diese Zahl ist verglichen mit der Anzahl an Wanderern recht gering, dies liegt [...] daran [...], dass in den Beispieldateien nicht realistische Werte angegeben sind, kein Mensch wandert an einem Tag mehr als zweimal um die Welt.

Ich werde nicht die Indizes der Teilnehmer ausgeben, da die Aufgabe von einer anonymen Umfrage spricht und es daher nicht sehr abgebracht ist, die Indices jedes Teilnehmers anzuzeigen.

In einer Implementierung der Idee in Python dauerte so ein Testfall (konkret `wandern7.txt`) ungefähr 70 Stunden (die Stromrechnung hat sich gefreut).

```
double gruppeDrei = Gruppenbilldung.gruppenbildungNachErsterGruppe(
StreckenNachGruppenbildungZwei, aT, differenzGeordnetNachGruppenbildungZwei,
uebrigeZwei);
```

Aufgabe 5: Das ägyptische Grabmal

Sollte man sich bereits im Grab befinden, [...]

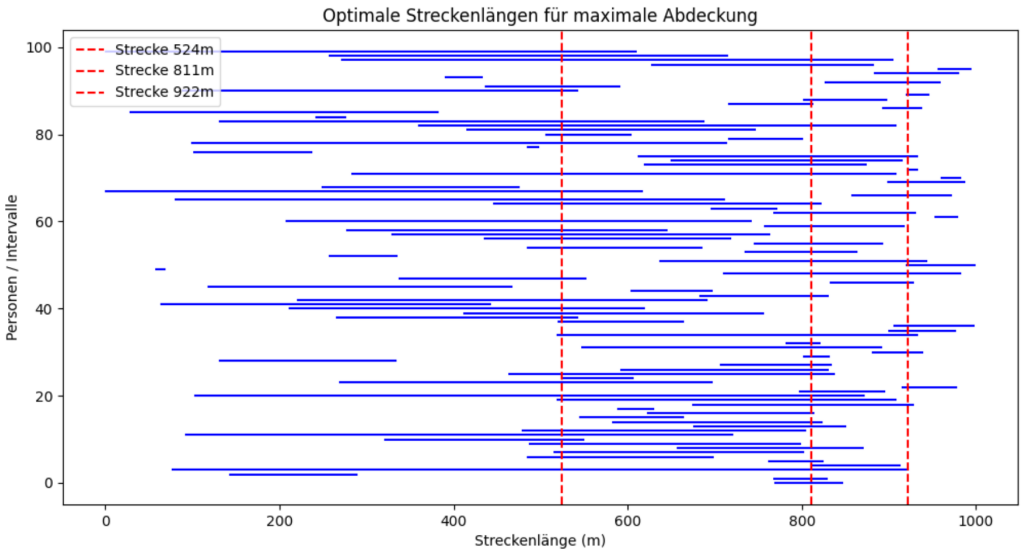
Dann hatte ich mit `grabmal5.txt` noch Probleme, da dies sehr kompliziert ist. Also habe ich einen Panic Mode eingebaut, welcher bewirkt, dass nicht mehr Schritte zurück gemacht werden.

Ich beginne bei 0 und gehe die Zeit entlang. Dann 'klone' ich Petra in alle möglichen Positionen.

Sie kann ihre Position beliebig ändern, aber die Zeit kann nur durch Warten erhöht werden.

Die Forscherin ist leider nach 40 Jahren im Labyrinth an Altersschwäche gestorben...
#21038400 Minuten sind 40 Jahre mit Berücksichtigung von Schaltjahren

Sie haben keinen Wert.



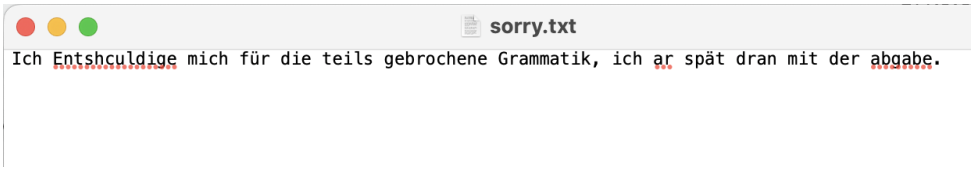
Man sieht, dass der Algorithmus die erste Linie bei 266 Meter setzt, denn hier sind viele Informatiker, die nicht so weit laufen wollen, die meisten allerdings wollen weiter laufen, haben aber dafür sehr enge Grenzen, wie weit sie laufen wollen, deshalb befinden sich die beiden anderen Streckenlängen nahe beieinander

Hier habe ich, anders als bei den anderen beiden Aufgaben nicht direkt online ein ähnliches oder gleiches Problem gefunden, dass schon gelöst wurde. Ich musste also kreativ werden.

Erst gegen Ende aufgefallen ist mir hingegen der Punkt, dass die Zeit vorwärts verlaufen muss.

```
if((int)gang.get(gang.size()-1).get(gang.get(gang.size()-1).size()-1)==1) {//Zelle offen
    gang.get(gang.size()-1).set(gang.get(gang.size()-1).size()-1, element: 2);
    durchsucht.get(gang.size()-1).set(gang.get(gang.size()-1).size()-1, element: true);
    ArrayList<Zelle> zellen = new ArrayList<>();
    zellen.add(new Zelle(gang.size()-1, gang.get(gang.size()-1).size()-1));
```

Kritisches Hinterfragen der eigenen Lösung



$unendlich = 10 * *20$

Leider funktioniert der Code aber aus mir unbekannten Gründen nicht, wenn man eines der Beispiele vom Bundeswettbewerb außer den ersten eingibt.

Leider kann ich aus mir unbekannten Gründen dieses Beispiel nicht lösen. Der Algorithmus funktioniert jedoch trotzdem.

Da der Algorithmus noch nicht richtig funktioniert, wird immer ausgegeben, dass keine Lösung vorhanden ist!

```
# Sicherheitsabbruch, falls etwas schiefgeht
```

Wegen Fehler, der eigentlich nicht existieren kann, nicht möglich.

Wunderschönes Format

Junior Aufgabe 2: Texthopsen

Blau = implementiert Rot = Beispiel

Die Idee:

Meine Idee wie das Programm Funktioniert war das das Programm für beide Spieler abwechselnd gemäß der Regeln zieht wer zuerst aus dem Text springt gewinnt .

Diese Idee habe ich wie folgt ungesetzt:

Beispieltext:

**Bela und Amira ist im Deutschunterricht oft lang-
weilig. Daher haben sie sich ein neues Spiel**