BUNDESWEIT INFORMATIK NACHWUCHS FÖRDERN
Bundeswettbewerb Informatik

Lösungshinweise

Allgemeines

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

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

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

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

Bewertungsbogen

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

Terminlage

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

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

Dokumentation

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

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

Einige unterhaltsame Formulierungsperlen sind im Anhang wiedergegeben.

Bewertung

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

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

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

Danksagung

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

An der Erstellung der im Folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Martin Schmidt (Aufgabe 1), Hans-Martin Bartram und Tim Pokart (Aufgabe 2) sowie Samuel Leßmann und Niels Glodny (Aufgabe 3). Allen Beteiligten sei für ihre Mitarbeit ganz herzlich gedankt.

Aufgabe 1: Weniger krumme Touren

1.1 Lösungsidee

Auf den ersten Blick erinnert die Aufgabenstellung an das bekannte Problem des Handlungsreisenden (Englisch: Traveling Salesman Problem, TSP). In beiden Fällen muss eine Reihe von Orten besucht werden, wobei die Länge der Reiseroute minimiert werden soll.

Es gibt aber auch Unterschiede. Während beim TSP die Reise meist am Ausgangsort wieder endet, ist dies in der Aufgabenstellung nicht gefordert. Auch kann man beim TSP typischerweise von jedem Ort zu jedem anderen Ort reisen. Und genau an dieser Stelle setzt die Aufgabe zentral an. Durch die Vorgabe eines maximalen Abbiegewinkels sind die möglichen auf eine Außenstelle folgenden Außenstellen eingeschränkt. Wie wir in Abschnitt 1.2 sehen werden, führt dies dazu, dass sich das Problem grundlegend ändert.

Während für das TSP jede Reihenfolge von Orten eine mögliche, erlaubte Route darstellt, ist bei dieser Aufgabe die Existenz einer Route, welche die Bedingung an den Abbiegewinkel erfüllt, nicht immer gegeben. Das TSP ist deshalb primär ein Optimierungsproblem, während bei dieser Aufgabe das Finden irgendeiner möglichen Route bereits ein Problem darstellt.

Dies spiegelt sich auch direkt in der Lösungsidee wieder. Eventuell gibt es Ansätze, die für viele Beispiele eine mögliche Route finden. Von praktischem Interesse ist aber auch, wie mit einem Beispiel umgegangen wird, in dem keine Route existiert. Wie kann nachgewiesen werden, dass es tatsächlich keine Lösung gibt? Das geht (mutmaßlich) nur durch Ausprobieren aller möglichen Routen.

Die Hoffnung ist, dass sich durch die Bedingung an den Abbiegewinkel die Anzahl der möglichen Routen soweit reduziert, dass wir eine Route iterativ aufbauen können. Im Grunde ist es ein Backtracking-Ansatz, der mit geeigneter Optimierung zum Erfolg führt.

1.2 Existenz einer Route

Als Erstes schauen wir uns an, ob es immer möglich ist, zu einer Liste von Punkten eine Route zu finden, die die Bedingungen an den Abbiegewinkel erfüllt.

Dass dies nicht der Fall ist, zeigen folgende Gegenbeispiele:

- Abbildung 1.1: Für Punkte, die in einem spitzwinkligen Dreieck angeordnet sind, ist es nicht möglich, eine Route anzugeben.

  Das Beispiel lässt sich aber auch noch um beliebig viele Punkte im Inneren des Dreiecks erweitern.

  Denn: Sobald man einen der drei Eckpunkte des Dreiecks auf einer Route besucht, endet die Route. Mit einem Abbiegewinkel von weniger als 90° ist kein Punkt des spitzwinkligen Dreiecks mehr erreichbar.

- Abbildung 1.2: Dieses Gegenbeispiel orientiert sich an `wenigerkrumm1.txt`, mit dem Unterschied, dass A und B die Start- und Zielpunkte der Route sein müssen, da der Abbiegewinkel bei A und B für alle möglichen Routen durch diese Punkte größer als 90° ist.

- Abbildung 1.3: Auch für das dritte Beispiel ist keine Route zu finden, die den Bedingungen an den Abbiegewinkel genügt. Ohne den Punkt A in der Mitte wäre hingegen eine Kreisroute durch alle Punkte möglich.

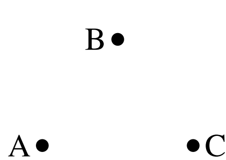


Abbildung 1.1: spitzwinkliges Dreieck

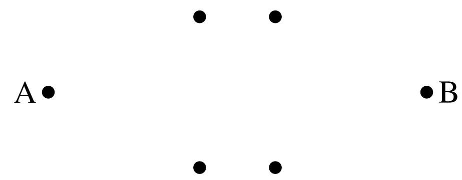


Abbildung 1.2: Pfade zwischen A und B

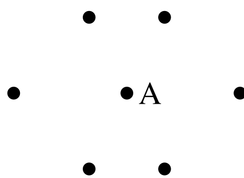


Abbildung 1.3: Punkt im Inneren

Wie wir in Abschnitt 1.1 gesehen haben, beeinflusst die Tatsache, dass es keine Route geben muss, die Lösungsidee maßgeblich.

1.3 Berechnung des Abbiegewinkels

Wenn wir von der Außenstelle A nach B geflogen sind, wollen wir prüfen, ob es möglich ist, weiter nach C zu fliegen. Dazu benötigen wir im Dreieck $\triangle ABC$ den Winkel β, siehe Abbildung 1.4.

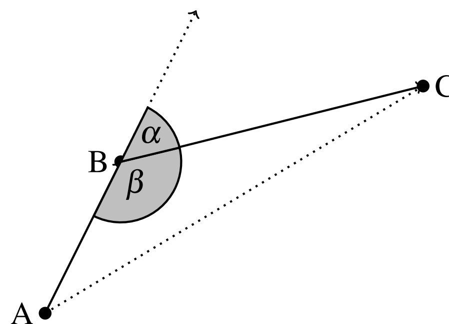


Abbildung 1.4: Skizze zur Berechnung des Abbiegewinkels

Damit ergibt sich der Abbiegewinkel α über

$$\alpha = 180^\circ - \beta.$$

Den Winkel β können wir über den Kosinussatz bestimmen:

$$\cos(\beta) = \frac{|\overline{AB}|^2 + |\overline{BC}|^2 - |\overline{AC}|^2}{2|\overline{AB}||\overline{BC}|}$$

Eigentlich wollen wir aber gar nicht den genauen Winkel bestimmen, sondern nur wissen, ob der Abbiegewinkel α größer oder kleiner gleich 90° ist. Dafür können wir uns eine Eigenschaft der Kosinus-Funktion zunutze machen: Der Kosinus hat eine Nullstelle bei 90° und ist negativ auf dem Intervall (90° , 180°].

Für $\beta \in [0°, 180°]$ gilt für den Abbiegewinkel α somit:

$$\alpha \leq 90° \text{ genau dann, wenn } \cos(\beta) \leq 0.$$

Wir müssen also nur

$$|\overline{AB}|^2 + |\overline{BC}|^2 - |\overline{AC}|^2$$

ausrechnen und das Vorzeichen bestimmen, um zu entscheiden, ob ein Abbiegewinkel erlaubt ist oder nicht.

1.4 Umsetzung

Eine vereinfachte Version des Algorithmus ist in Algorithmus 1 beschrieben. Wir starten mit einer leeren Route und fügen iterativ weitere Punkte hinzu. Dazu prüfen wir den Abbiegewinkel für alle möglichen Flugziele und rufen den Algorithmus rekursiv mit einer um einen Punkt verlängerten Route auf. Sollte es zu einem gewählten Punkt keine Lösung geben, kehrt der Aufruf ohne Ergebnis zurück und der nächste Punkt kann ausprobiert werden.

Optimierungen

Mit Algorithmus 1 lassen sich nicht für alle Beispieleingaben mögliche Routen finden. Zum Glück haben wir noch Raum für Optimierungen.

- Die möglichen nächsten Zielorte (Algorithmus 1, Zeilen 6-15) hängen nur von den beiden zuletzt besuchten Orten/Punkten ab. Somit lassen sich mögliche Zielorte in Abhängigkeit von den beiden vorherigen Orten vorab berechnen oder auch cachen, um diese nur einmal berechnen zu müssen.
- Die Reihenfolge, in der diese nächsten Zielorte ausprobiert werden, hat einen entscheidenden Einfluss auf das schnelle Finden einer möglichen Lösung. Wenn wir eine möglichst kurze Route finden wollen, sollten diese Zielorte nach dem aufsteigenden Abstand zum zuletzt besuchten Ort sortiert werden.
- Die Wahl der Anfangspunkte ist im Prinzip zufällig. Jeder Punkt könnte potentiell der Anfang einer möglichen Lösung sein. Um die Ressourcen moderner Hardware besser auszunutzen, ist es möglich, Algorithmus 1 nicht mit einer leeren Liste aufzurufen, sondern

  $$\text{ROUTE_FINDEN}([p]) \qquad \text{für } p \in P$$

  gleichzeitig zu berechnen. Als hilfreich hat sich ein Zeitlimit für die einzelnen Startpunkte erwiesen, da die Anzahl der Punkte zu groß ist, um tatsächlich alle gleichzeitig zu berechnen, aber nur für einige wenige Startpunkte tatsächlich eine Lösung in annehmbarer Zeit gefunden werden kann. Je nach Hardware reicht ein Zeitlimit von 1s pro Startpunkt aus, um für alle Beispieleingaben eine mögliche Route zu finden.

Algorithmus 1 Vereinfachter Algorithmus zum Finden einer Lösung

```
Require: P - Liste aller Punkte
 1: procedure ROUTE_FINDEN(path)                          ▷ path: Liste besuchter Punkte
 2:     if length(path) = length(P) then
 3:         mögliche Route path gefunden
 4:         Programm beenden und path ausgeben
 5:     end if
 6:     destinations ← []                          ▷ mögliche nächste Zielorte bestimmen
 7:     for all p ∈ P do
 8:         if length(path) ≤ 1 then                  ▷ keine Abbiegebedingung anwendbar
 9:             destinations ← destinations + [p]
10:         else
11:             if angle(path[−2], path[−1], p) ≥ 90° then          ▷ siehe Abschnitt 1.3
12:                 destinations ← destinations + [p]
13:             end if
14:         end if
15:     end for
16:     for all p ∈ destinations do
17:         if p ∉ path then
18:             ROUTE_FINDEN(path + [p])                              ▷ Rekursiver Aufruf
19:         end if
20:     end for
21:     return                ▷ Backtracking-Schritt, falls keine Lösung gefunden wurde
22: end procedure
```

Um nicht nur eine mögliche Route zu finden, sondern eine möglichst kurze Route, können wir weitere Anpassungen an Algorithmus 1 vornehmen:

- Anstatt nach dem Finden der ersten Lösung in Zeile 4 abzubrechen, können wir die gefundene Lösung zurückgeben und in Zeile 18 dann das Minimum über alle gefundenen Routen bilden.
- Wenn wir die bisher beste gefundene Route tracken, kann deren Länge auch als Abbruchkriterium genutzt werden: Wenn die aktuell untersuchte Route zu lang ist, kann das weitere Verlängern der Route abgebrochen werden.

Die tatsächlich kürzeste Route lässt sich damit für die Beispieleingaben immer noch nicht in allen Fällen in angemessener Zeit bestimmen. Die besten gefundenen Routen sind in Abschnitt 1.6 gelistet.

1.5 Laufzeitbetrachtung

Eine Laufzeitbetrachtung für den vorgestellten Algorithmus stellt sich als schwierig dar.

Wenn wir mit n die Anzahl der Punkte bezeichnen, gibt es ohne die Einschränkung des Abbiegewinkels $\mathcal{O}(n!)$ mögliche Routen: In jedem Schritt stehen alle noch nicht besuchten Punkte zur Verfügung.

Verbessern kann ich diesen Wert mit Hilfe des Kriteriums an den Abbiegewinkel aber nicht.

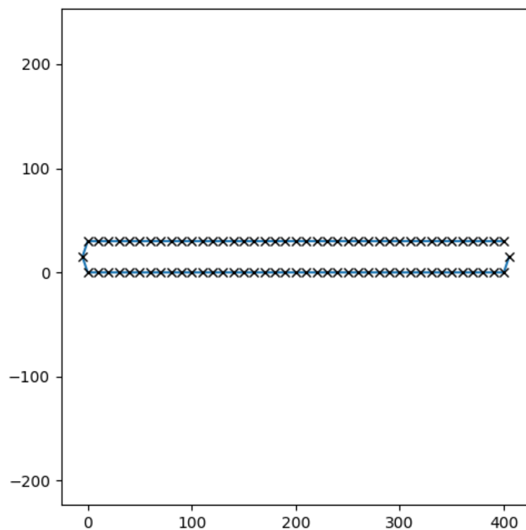
In der Praxis lassen sich Routen mit deutlich weniger Aufwand berechnen, d. h. es werden oft bei weitem nicht alle $n!$ Möglichkeiten tatsächlich probiert.

1.6 Beispiele

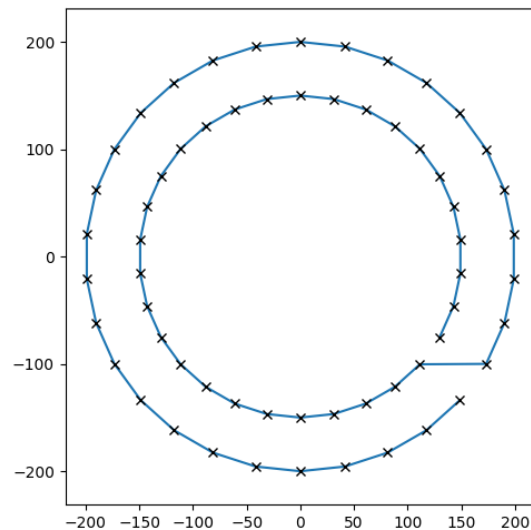
Für die vorgegebenen Beispiele findet das Programm folgende Routen.

Dabei ist zu beachten:

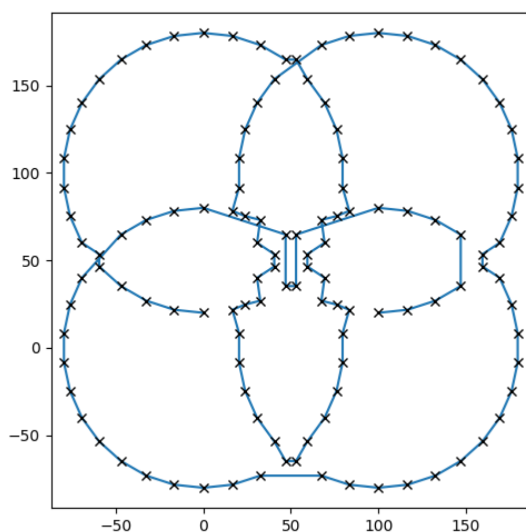
- Die Routenlängen sind auf eine Nachkommastelle gerundet.
- Dies sind nicht notwendigerweise die kürzesten Routen.



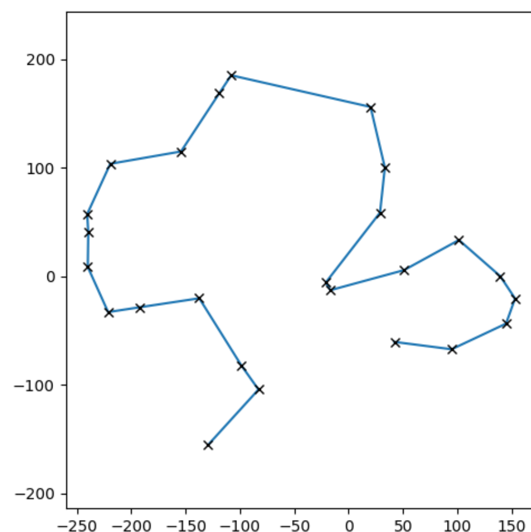
`wenigerkrumm1.txt`
Routenlänge: 847,4



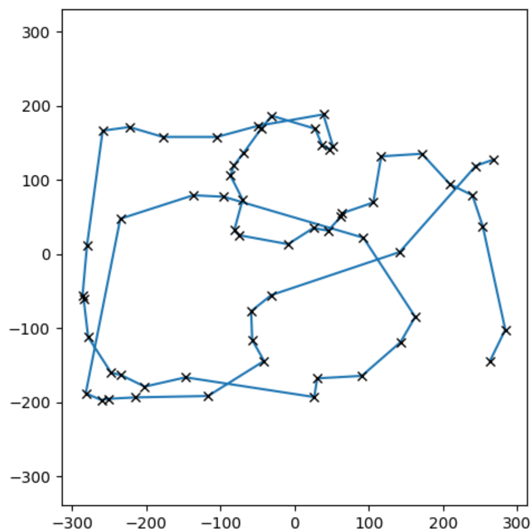
`wenigerkrumm2.txt`
Routenlänge: 2183,7



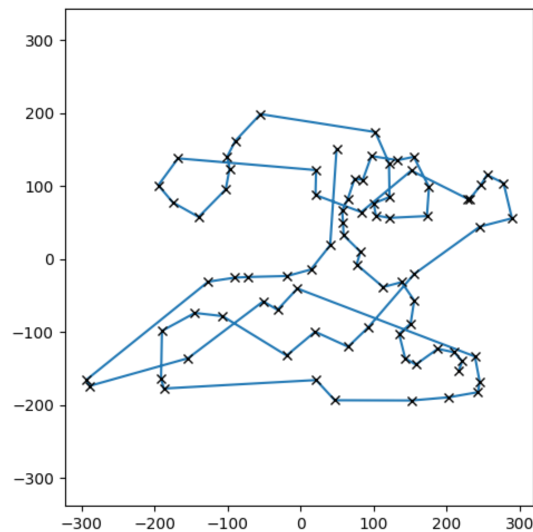
`wenigerkrumm3.txt`
Routenlänge: 1936,0



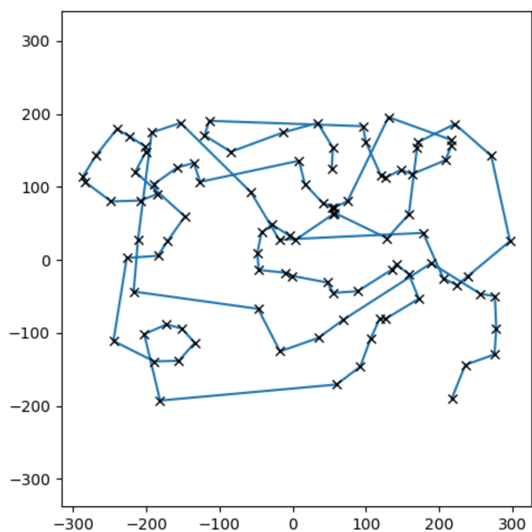
`wenigerkrumm4.txt`
Routenlänge: 1205,1



`wenigerkrumm5.txt`
Routenlänge: 3530,2



`wenigerkrumm6.txt`
Routenlänge: 4146,8



`wenigerkrumm7.txt`
Routenlänge: 5055,0

Mit anderen Verfahren lassen sich für die Eingabebeispiele die folgenden optimalen Routenlängen bestimmen:

Beispieleingabe	Routenlänge [km]
`wenigerkrumm1.txt`	847,43
`wenigerkrumm2.txt`	2183,66
`wenigerkrumm3.txt`	1848,05
`wenigerkrumm4.txt`	1205,07
`wenigerkrumm5.txt`	3257,92
`wenigerkrumm6.txt`	3457,99
`wenigerkrumm7.txt`	4150,64

1.7 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Jede sinnvolle Modellierung des Problems ist erlaubt. Die Modellierung darf die Suche nach einer möglichst kurzen Route und die korrekte Bestimmung des Abbiegewinkels nicht unnötig verkomplizieren.
(2) *Verfahren nicht unnötig ineffizient*: In die ohnehin aufwändige Lösungssuche sollen unerlaubte Touren (mit unerlaubten Abbiegewinkeln) nicht grundlos einbezogen werden. Wenn etwa alle Permutationen von Punkten aufgezählt und für jede die Abbiegewinkel überprüft werden, werden 4 Punkte abgezogen.
(3) *Laufzeit des Verfahrens in Ordnung*: Eine erste mögliche Route sollte innerhalb kurzer Zeit für alle vorgegebenen Beispiele gefunden werden können. Für die Optimierung der Routenlänge ist dann auch eine längere Laufzeit in Ordnung, sofern sie zu einer Verkürzung der Routenlänge führt. Können nur sehr kleine Beispiele gelöst werden, werden bis zu 4 Punkte abgezogen; werden nicht alle Pflichteingaben gelöst, dann eher 2 Punkte. Verfahren, die vielleicht nur mit etwas Glück alle Beispiele irgendwie lösen können (und in der Regel dann schlechte Ergebnisse liefern), werden bei diesem Punkt eher gnädig behandelt; Abzüge sind in solchen Fällen an anderer Stelle möglich.
(4) *Speicherbedarf in Ordnung*: Da die Beispieleingaben mit maximal 120 Punkten / Orten eine eher kleine Eingabegröße haben, sollte sich der benötigte Speicher in Grenzen halten.
(5) *Verfahren mit korrekten Ergebnissen*: Die generierten Routen müssen alle Punkte einbeziehen und die Bedingung an den Abbiegewinkel einhalten. Die Route kann an beliebigen Orten starten und enden. Wenn die gefundenen Routen kürzer sind als die angegebenen optimalen Routenlängen, kann davon ausgegangen werden, dass eine Bedingung (im Zweifel die an die Abbiegewinkel) nicht korrekt eingehalten wurde.
(6) *Verfahren mit guten Ergebnissen*: Die Routenlängen sollen ungefähr denen des beschriebenen approximativen Verfahrens entsprechen. Bei schlechteren Ergebnissen können bis zu 4 Punkte abgezogen werden – insbesondere auch dann, wenn die Bearbeitung sich eher nicht um Optimierung der Routenlänge bemüht. Wenn für mindestens zwei der Beispieleingaben ab `wenigerkrumm5.txt` bessere Ergebnisse erreicht werden, so können 2 Pluspunkte vergeben werden. Das gilt aber nur, wenn eigene Optimierungsideen zum Einsatz kommen; die Verwendung fertiger Tools, etwa ILP-Solver, soll hier nicht belohnt werden.

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Es muss insbesondere erkannt werden, wenn das verwendete Verfahren keine optimalen Ergebnisse liefert – die ja auch nicht gefordert waren. Genau so sollte erkannt werden, wenn ein (einfaches) Verfahren potenziell schwache Ergebnisse liefern kann.
(2) *Mögliche Unlösbarkeit erkannt*: Es muss erkannt werden, dass nicht zwangsläufig für alle Eingaben eine Lösung existiert; ansonsten werden 2 Punkte abgezogen. Ein Beispiel für eine nicht lösbare Eingabe ist schön, aber noch keine besondere Leistung. Werden

unlösbare Fälle mit einem gewissen Aufwand verallgemeinert charakterisiert, kann das mit Pluspunkten belohnt werden.

(3) *Gute Überlegungen zur Laufzeit des Verfahrens*: Die Laufzeit kann nur schwer abgeschätzt werden. Praktische Laufzeitmessungen können die Angabe der asymptotischen Laufzeit ersetzen. In Kombination – insbesondere, wenn die tatsächliche Laufzeit auf den Beispielen stark von der Worst-Case-Laufzeit abweicht – kann eine gründliche Analyse des Unterschieds zwischen asymptotischer und tatsächlicher Laufzeit auch Pluspunkte geben.

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Es sollten alle vorgegebenen Beispiele bearbeitet und dokumentiert sein. Wenn nur etwa die Hälfte dokumentiert ist, gibt es 2 Punkte Abzug.

(5) *Ergebnisse nachvollziehbar dargestellt*: Da es insbesondere um die Länge der Routen geht, muss diese angegeben werden; ansonsten wird 1 Punkt abgezogen. Es muss nachprüfbar sein, ob der maximale Abbiegewinkel von 90° tatsächlich eingehalten wird. Dies ist insbesondere durch eine grafische Darstellung der Routen möglich. Unterschiedliche Skalierungen der Achsen sind dabei nicht sinnvoll, weil rechte Winkel dann nicht erkannt werden können. Da dies häufig nicht bedacht wurde, soll auf Punktabzug aber verzichtet werden.
Es ist auch akzeptabel, wenn die Route als Text ausgegeben und der Abbiegewinkel in jedem Schritt angegeben wird.

Aufgabe 2: Alles Käse

2.1 Lösungsidee

Noch in Gedanken an den leckeren Käse versunken kommt einem möglicherweise zunächst eine ganz naive Idee zur Lösung in den Sinn: Hat man alle n Scheiben $S = (s_1, \dots, s_n)$ gegeben, so kann man beginnend mit der ersten alle als Startscheibe ausprobieren und jeweils versuchen, die verbleibenden Scheiben anzulegen. So soll der Käse von innen nach außen zusammen gesetzt werden. Aber die Startscheibe ist dabei nicht notwendigerweise immer die kleinste Scheibe!

Allgemein gilt, dass eine Scheibe s mit den Seitenlängen a und b an einen Käse K mit den Seitenlängen x, y und z angelegt werden kann, wenn zwei der drei Käseseiten den Scheibenseitenlängen entsprechen.[1] Die dritte Käseseite wächst durch das Anlegen der Scheibe um einen Millimeter. Wird durch das wiederholte Anlegen ein Zustand erreicht, in dem keine anzulegenden Scheiben mehr vorhanden sind, ist der Käse rekonstruiert.

Solch ein Programm hat eine Laufzeit von $\mathcal{O}(n!)$ und ist damit voraussichtlich nicht geeignet, um Antjes große Käseblöcke zusammenzusetzen. Nichtsdestotrotz bietet es eine gute Grundlage, die wir im Folgenden verbessern werden.

Eine offensichtliche Verbesserungsmöglichkeit bieten gleich große Scheiben. Wenn zwei Scheiben s und s' die gleichen Seitenlängen haben, nutzt es nichts, beide als Startkäsescheibe oder an ein und derselben Stelle als Anlegescheibe zu probieren. Stattdessen reicht es aus, einen Vertreter dieser identischen Scheiben auszuprobieren. Das motiviert, die Menge $S^* = \{ s_i^* \}$ der einzigartigen Käsescheiben zu definieren, worin von jeder Käsescheibengröße nur ein Vertreter s^* auftaucht. Die Anzahl einzigartiger Scheiben nennen wir $n^* = |S^*|$.

Die nächste Verbesserungsmöglichkeit betrifft die Überprüfung, ob eine Scheibe überhaupt an den Käse angelegt werden kann. Betrachten wir wieder einen Käse mit den der Größe nach sortierten Seitenlängen $x \leq y \leq z$. Zum Anlegen kommen hier ausschließlich Scheiben mit den Längen x, y sowie x, z und y, z in Frage. Weiterhin lässt sich feststellen, dass die Scheibe x, y unbedingt angelegt werden muss, insofern sie existiert. Würde man stattdessen eine andere Scheibe anlegen, wäre entweder die Seite x oder die Seite y einen Millimeter größer und man hätte nie wieder die Möglichkeit die Scheibe x, y anzulegen, weil es eben die zwei kleinsten Seiten waren. Existiert diese nicht, muss nach der gleichen Argumentation die Scheibe x, z angelegt werden, da beim Anlegen der sonst verbleibenden Scheibe y, z die Seite x um einen Millimeter wachsen würde und die Scheibe x, z nie wieder angelegt werden könnte. Existiert eine solche Scheibe ebenfalls nicht, muss die letzte verbleibende Scheibe y, z angelegt werden. Werden nach dieser Reihenfolge alle Scheiben angelegt, so ist der Käse wieder zusammengesetzt. Sollte einmal keine passende Scheibe gefunden werden, so kann der Käse von der gewählten Startscheibe aus nicht zusammengesetzt werden. Da ausgehend von jedem Startblock die n folgenden Scheiben durch diese Reihenfolge eindeutig bestimmt sind, ergibt sich hier eine Laufzeit von $\mathcal{O}(nn^*)$ unter der Annahme, dass die Existenz einer Scheibe in $\mathcal{O}(1)$ festgestellt werden kann.

In Algorithmus 2 ist ein Pseudocode für eine rekursive Umsetzung unter Berücksichtigung der oben genannten Verbesserungen gegeben. Um die Beispieleingaben von BWINF lösen zu können, muss teilweise von der rekursiven Struktur von Algorithmus 2 zu einer iterativen Umsetzung gewechselt werden, damit auch sehr große Scheibenmengen behandelt werden können.

[1] Genau genommen macht es für den finalen Käse einen Unterschied, ob man eine Scheibe jeweils oben oder unten, links oder rechts, oder vorne oder hinten anlegt. Allerdings ist zunächst nur nach *einer* Möglichkeit gefragt, den Käse zusammenzusetzen, weswegen dieses technische Detail hier praktisch nicht weiter beachtet wird.

In den meisten Programmiersprachen würde die Rekursion nämlich zu einem Stapelüberlauf (Stackoverflow) führen.

Algorithmus 2 Käserekonstruktionsprogramm

Input: Menge S von Scheiben

```
 1: for each einzigartige Scheibe s* in S* do
 2:     Konstruiere Startkäse K_0 = s*
 3:     Markiere s* als verwendet
 4:     VERSUCHE ANZULEGEN(Käse K_0)
 5: end for

 6: procedure VERSUCHE ANZULEGEN(Käse K)
 7:     if alle Scheiben verwendet then
 8:         Eine Möglichkeit zur Käserekonstruktion wurde gefunden
 9:     end if
10:     Finde nach der Anlegereihenfolge an Käse K passende Scheibe s*
11:     if Scheibe s* existiert then
12:         Lege s* an Seite von K an und erhalte K'
13:         Markiere s* als verwendet
14:         VERSUCHE ANZULEGEN(Käse K')
15:     else
16:         Käse ist nicht rekonstruierbar
17:     end if
18: end procedure
```

Volumentrick

Algorithmus 2 kann noch weiter verbessert werden. In Zeile 2 muss für jede einzigartige Scheibe überprüft werden, ob diese als möglicher Startkäse dienen kann. Mit der folgenden Beobachtung kann allerdings eine große Zahl von Startscheiben ausgeschlossen werden:

Lässt sich der Käse K vollständig rekonstruieren, hat dieser das Volumen V_K, welches sich aus den Volumina $V(s_i)$ aller ursprünglichen Käsescheiben s_i als Summe $V_K = \sum_{i=0}^{n} V(s_i)$ ergibt. Hat dieser Käse am Ende die Maße x, y und z, so ist $V_K = x * y * z$. Angenommen, die letzte Scheibe wurde in z-Richtung angelegt[2], dann hatte die letzte Scheibe s die Maße x und y und das Volumen $V(s) = x * y$, womit schlussendlich $V_K = V(s) * z$ folgt. Es kommen also nur solche Scheiben als letzte Scheibe in Frage, deren Volumina Teiler des Gesamtkäsevolumens sind. Damit können sehr effizient die möglichen Kandidaten für eine letzte anzulegende Käsescheibe eingegrenzt werden.

Der Käse kann so nicht wie vorher von einer Startscheibe beginnend von innen nach außen, sondern muss beginnend mit einem möglichen Kandidaten für die letzte anzulegende (also zuerst abgeschnittene) Scheibe von außen nach innen rekonstruiert werden. Der dies umsetzende Algorithmus ist bis auf wenige Details sehr ähnlich zu Algorithmus 2, sogar die Betrachtungen zur Anlegereihenfolge können in umgekehrter Reihenfolge wiederverwendet werden. Anstatt

[2]Die Argumentation ist für ein Anlegen in jeweils x- und y-Richtung äquivalent. Die Einschränkung auf z ist daher ohne Beschränkung der Allgemeinheit.

eine Scheibe anzulegen, müssen hier nun Scheiben vom Käse abgeschnitten werden. In Algorithmus 3 ist die entsprechende Anpassung der Auswahl möglicher Startscheiben angegeben.

Algorithmus 3 Käserekonstruktionsprogramm mit Volumentrick

Input: Menge S von Scheiben

1: Berechne das finale Käsevolumen $V = \sum_i V(s_i)$
2: **for each** einzigartige Scheibe $s^* = (a,b)$ in S^* **do**
3: Überspringe s^*, wenn $V(s^*) = a*b$ kein Teiler von V
4: Konstruiere Startkäse K_0 mit Seitenlängen $V/V(s^*)$, a, b
5: Markiere s^* als verwendet
6: VERSUCHE ABZUSCHNEIDEN(Käse K_0)
7: **end for**

Datenstrukturen Die Scheiben können als geordnete Paare ganzer Zahlen $s = (a,b)$ dargestellt werden, wobei beispielsweise die kleinere Seite zuerst genannt wird. Die Ordnung der Scheibe bietet sich an, um schnell überprüfen zu können, ob es eine zur Käseseite x und y passende Scheibe gibt. Dafür können eine Vielzahl von Datenstrukturen verwendet werden, die allesamt ein promptes Überprüfen der Existenz einer Scheibe mit den jeweiligen Maßen und der von dieser Scheibenart noch verfügbaren Anzahl ermöglichen sollen. Einige Beispiele dafür:

- In einem Array wird jeder möglichen Länge der kurzen Seite einer Scheibe eine Liste vorhandener langer Seiten zuordnet. Findet sich der entsprechende Eintrag, kann so schnell überprüft werden, ob eine Scheibe einer bestimmten Größe existiert.
- Alternativ kann eine Map verwendet werden, bei der jedem Scheibenhash die verbleibende Scheibenanzahl zugeordnet wird. Ist der Hash in der Map vorhanden, kann die Existenz einer Scheibe überprüft werden.

Empirisch konnte festgestellt werden, dass Implementierungen, in denen die Map als Datenstruktur verwendet wurde, im Abtausch eines leicht erhöhten Speicherbedarfs bedeutende Laufzeitvorteile erzielen.

Um die iterative Lösung umzusetzen, wird ein Stapel zum Mitschreiben der Rücksprungpunkte verwendet. Für die Käsegröße kann ein Array mit Ganzzahlen verwendet werden.

Modellierung als Graph Eine andere Sicht auf die Aufgabe bekommt man, wenn man die Scheibenmenge als Graph modelliert. Fügt man in diesen für jede einzigartige Scheibe einen Knoten ein und verbindet Knoten, die mindestens eine Seitenlänge teilen, ergibt sich ein Graph wie in Abbildung 2.1. In diesem können nun alle Knoten als Startscheibe (mit etwaigen Einschränkungen wie vorher) ausprobiert werden und durch eine Suche auf dem Graphen die möglichen Anlegescheiben bestimmt werden. Für das direkte Anlegen kommen nämlich nur die Scheiben bzw. Knoten in Frage, die mit der aktuellen mindestens eine Seitenlänge gemeinsam haben, ergo im Graph verbunden sind.

In dieser Darstellung wird eine weitere Optimierung für die Auswahl der Startscheiben augenfällig, und zwar aus der folgenden Beobachtung: Angenommen die Scheibe $(2,4)$ wird als Startscheibe in Betracht gezogen. Dann können nachfolgend nur die Scheiben $(3,4)$ und $(4,6)$ oder $(2,4)$ selbst angelegt werden, weil sie sich ja mindestens eine Seite mit der Startscheibe teilen müssen. Damit es nun aber möglich ist, überhaupt eine andere als die Startscheibe

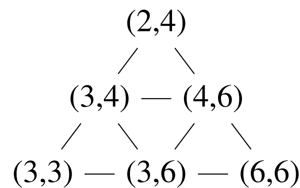


Abbildung 2.1: Darstellung von `kaese1.txt` als Graph.

anzulegen, muss diese mindestens so oft gestapelt werden, wie die kürzeste Seite der mit ihr verbundenen Scheiben (oder ihre kürzeste Seite selbst) lang ist. Sind weniger Exemplare als die kürzeste Seite der Scheibe oder ihrer Nachbarn vorhanden, kann diese direkt als Startscheibe ausgeschlossen werden. Dieses Kriterium kann noch etwas gelockert werden, in dem man fordert, dass mindestens so viele Exemplare wie die kürzeste Seite *aller* Scheiben vorhanden sein muss.

2.2 Erweiterungen

Abhängig von Antjes Käsevorlieben und Verhalten während des Schneidens sind einige Erweiterungen denkbar, wobei hier die in der Aufgabenstellung vorgeschlagenen Erweiterungen diskutiert werden.

Verschwundene Scheiben Einerseits besteht die Möglichkeit, dass Antje während des Telefonats bereits einige Scheiben genascht hat und der Käse dadurch mit einer unvollständigen Menge an Scheiben vervollständigt werden muss. Wenn Scheiben verschwunden sind, kann der Volumentrick aus Algorithmus 3 nicht mehr verwendet werden, weil das endgültige Käsevolumen V schlichtweg nicht bekannt ist. Stattdessen muss eine angepasste Version von Algorithmus 2 verwendet werden. Bei dieser können zusätzlich M fehlende Scheiben zu S ergänzt werden, wobei die Beschränkung auf eine Obergrenze fehlender Scheiben essentiell ist. Existiert zum Anlegen an eine Seite keine Scheibe, so wird versucht, den Käse durch das Anlegen einer fiktiven Scheibe zu rekonstruieren.

Fehlen mehr als eine Scheibe, so können die Verbesserungen aus der Anlegereihenfolge nicht mehr verwendet werden.[3] Nun muss für jede Seite des Käses überprüft werden, ob eine anlegbare Scheibe existiert, wodurch die Laufzeit exponentiell in der Scheibenanzahl skaliert. Dadurch werden Optimierungen, wie die Berücksichtigung von gleichen Käseseiten, bei denen der gleiche Suchbaum nicht wiederholt abgelaufen werden muss, notwendig.

Mehrere ursprüngliche Käseblöcke Hatte Antje mehrere Käsesorten für ihr Brot vorgesehen und die Scheiben ununterscheidbar[4] beim Schneiden gemischt, müssen mehrere Käseblöcke rekonstruiert werden. Entstammen die Scheiben S allerdings mehreren Käseblöcken, wird

[3] Fehlt nur eine Scheibe, muss weiterhin die Anlegereihenfolge gelten. Gilt sie einmal nicht, kann eine fiktive Scheibe für jede Seite eingeführt werden, für alle nachfolgenden Schritte muss sie dann aber wieder gelten, weil die fehlende Scheibe bereits wiederhergestellt ist. Dadurch ist die Laufzeit weiterhin $\mathcal{O}(nn^*)$.

[4] Wenn Antje die Scheiben nicht anhand ihrer äußerlichen Merkmale außeinanderhalten kann, können dementsprechend auch die ursprünglichen Käse nicht garantiert rekonstruiert werden. Lediglich eine Zuordnung der Art, dass für eine paarweise disjunkte Aufteilung der Scheiben $S_1 \subset S, \dots, S_n \subset S$ (mit $S_i \cap S_j = \emptyset$ für $i \neq j$) die Käsequader $K_1, \dots, K_n$ zusammengesetzt werden können, ist möglich.

Algorithmus 3 ebenfalls unmöglich, da die Teilerbedingung nicht mehr erfüllt sein muss. Allerdings kann auch hier Algorithmus 2 so ergänzt werden, dass mehrere Käseblöcke bedacht werden. Zunächst startet man mit allen möglichen Kombinationen von N Startscheiben und überprüft zusätzlich die Möglichkeit an jeden der Käse eine Scheibe anzulegen. Hatte man vorher bei einem der Käse keinen Erfolg eine Seite anzulegen, so wird dies auch fortlaufend nicht gelingen. Der Käse kann somit zunächst für folgende Schritte gesperrt werden. Genauso müssen gleichen Seiten nicht noch einmal überprüft werden.

In dieser Erweiterung erhöht sich die Laufzeit auf $\mathcal{O}(\binom{n}{N}(3N)^n)$, weil für jede der $\binom{n}{N}$ Startpaarungen die $(3N)^n$ Anlegestellen versucht werden.

Algorithmus 4 zeigt, wie Algorithmus 2 um die beiden oben genannten Erweiterungen ergänzt werden kann.

Algorithmus 4 Erweitertes Käserekonstruktionsprogramm

Input: Menge S von Scheiben, zu rekonstruierende Quader N, fehlende Scheiben M

1: **for each** einzigartige Scheibenpaarung $\{s_1^*, \dots, s_N^*\}$ in S^* **do**
2: Konstruiere Startkäse $K_1, \dots, K_N$ von Scheiben $s_1^*, \dots, s_N^*$
3: Markiere $s_1^*, \dots, s_N^*$ als verwendet
4: VERSUCHE ANZULEGEN(Käse $K_1, \dots, K_N$)
5: **end for**

6: **procedure** VERSUCHE ANZULEGEN(Käse $K_1, \dots, K_N$)
7: **if** alle Scheiben verwendet **then**
8: Eine Möglichkeit zur Käserekonstruktion wurde gefunden
9: **end if**
10: **for each** nicht gesperrten Käses K_i **do**
11: **for each** Seite des Käses K_i **do**
12: Überspringe Seite, wenn ähnliche bereits überprüft
13: **if** Scheibe s^* passt an Seite des Käses K_i **and** nicht alle s^* verwendet **then**
14: Lege s^* an Seite von K_i an und erhalte K_i'
15: Markiere Vertreter von s^* als verwendet
16: VERSUCHE ANZULEGEN(Käse $K_1, \dots, K_i', \dots, K_N$)
17: **else if** weniger als M Zusatzscheiben verwendet **then**
18: Erschaffe Zusatzscheibe und lege sie an Seite von K_i an, um K_i' zu erhalten
19: VERSUCHE ANZULEGEN(Käse $K_1, \dots, K_i', \dots, K_N$)
20: **end if**
21: **end for**
22: Sperre Käse K_i ▷ An diesen Käse können keine Scheiben mehr angelegt werden
23: **end for**
24: Entsperre alle gesperrten Käse K_i
25: **end procedure**

Wurden nur zwei verschiedene Käse miteinander gemischt, ohne dass Scheiben genascht wurden, können Algorithmus 2 und Algorithmus 3 kombiniert werden, um wie in Algorithmus 5 dargestellt die beiden ursprünglichen Käse zu rekonstruieren. Dadurch ergibt sich im Vergleich zur allgemeinen Lösung ein Vorteil, weil Algorithmus 3 schnell die Nichtexistenz einer Lösung durch die Teilerbedingung feststellen kann.

Algorithmus 5 Käseprogramm für zwei gemischte Käse

Input: Menge S von Scheiben

1: Beginne den ersten Käse mit Algorithmus 2 aufzubauen
2: Bei jedem Anlegeschritt: Versuche die restlichen Scheiben mit Algorithmus 3 zusammenzusetzen

2.3 Beispiele

Mit den oben genannten Algorithmen kann man für jede Beispieldatei einen Käse konstruieren. Die Ergebnisse sind in Tabelle 1 gegeben. Für jede Datei sind jeweils die Maße der ersten und letzten vier angefügten Käsescheiben sowie die Maße des entstandenen Käsequaders angegeben.

Beispieldatei	Erste Scheibe, ..., Letzte Scheibe	Käsedimensionen
`kaese1.txt`	2×4, 2×4, 2×4, 3×4, ..., 4×6, 4×6, 6×6, 6×6	$6\times6\times6$
`kaese2.txt`	998×999, 998×999, 2×998, 2×1000, 2×1000	$1000\times2\times1000$
`kaese3.txt`	992×995, 992×995, 2×995, 2×993, ..., 7×1000, 1000×1000, 1000×1000, 1000×1000	$10\times1000\times1000$
`kaese4.txt`	29×51, 29×51, 29×51, 3×29, ..., 209×209, 209×209, 209×210, 210×210	$210\times210\times210$
`kaese5.txt`	437×1325, 437×1325, 437×1325, ..., 2308×3569, 2730×3569, 2309×2730, 2730×3570	$2310\times2730\times3570$
`kaese6.txt`	9255×480255, 9255×480255, 2×480255, 9256×480255, ..., 39269×510509, 30029×510509, 30029×39270, 39270×510510	$30030\times39270\times510510$
`kaese7.txt`	665×962, 665×962, 2×962, 2×962, ..., 510509×510510, 510508×510510, 510510×510510, 510510×510510	$510510\times510510\times510510$

Tabelle 1: Ergebnisse der verschiedenen Beispieldateien. Für jeden Käsewürfel sind jeweils die ersten und letzten vier angelegten Scheiben sowie die am Ende entstandene Käselänge angegeben.

Um die Methoden für vermischte Käse zu testen, können mehrere Beispieldateien vermischt werden. Dabei sollten paarweise Mischungen von `kaese1.txt` bis `kaese6.txt` möglich sein.

Um die Methoden für fehlende Käsescheiben zu testen, können aus den Beispieldateien zufällig Scheiben entfernt werden. Mit einer fehlenden Scheibe sollten noch alle Beispieldateien lösbar sein. Bei mehr fehlenden Scheiben werden die großen Eingabedaten zu schwer; mit fünf fehlenden Scheiben beispielsweise konnten nur noch bis `kaese5.txt` Lösungen gefunden werden.

2.4 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Die Scheiben sowie Käseseiten sollten geeignet, wenn möglich als Ganzzahlpaar, dargestellt werden. Genauso sollte der Käse als Ganzzahltripel dargestellt werden. Es muss eine sinnvolle Datenstruktur zum Erkennen der verfügbaren Scheiben gewählt werden.

(2) *Verfahren nicht unnötig ineffizient*: (Teil a:) Offensichtliche Unmöglichkeiten sollen bei den Versuchen zur Rekonstruktion des Käses berücksichtigt werden. Wenn alle Scheibenpermutationen aufgezählt werden und für jede das Zusammenlegen ausprobiert wird, ohne weitere Verbesserungen vorzunehmen, werden 4 Punkte abgezogen.

(3) *Laufzeit des Verfahrens in Ordnung*: Alle Beispieldateien sollten in weniger als einer Minute bearbeitet werden können. Für Verfahren mit polynomieller Laufzeit ist das problemlos möglich. Sollten andere Verfahren nur die Beispiele `kaese1.txt` bis `kaese5.txt` erfolgreich verarbeiten können (z. B. weil nicht ausreichend optimiert wurde), gibt es 2 Punkte Abzug; bis zu 4 Punkte werden abgezogen, wenn nur die kleinsten Beispiele gelöst werden. Sind mehr als drei sinnvolle und effektive Optimierungen angegeben und umgesetzt bzw. können noch deutlich größere Eingaben gelöst werden, kann es bis zu 2 Pluspunkte geben.

(4) *Speicherbedarf in Ordnung*: Der Speicherbedarf sollte insgesamt, also für Scheiben, Käse und Anlegestapel, nicht bedeutend schlechter als linear in der Scheibenanzahl sein. Wird ein Graph zur Abhängigkeitsdarstellung gewählt, sollte dieser nicht als Adjazenzmatrix kodiert sein.

(5) *Verfahren mit korrekten Ergebnissen*: Das Verfahren muss den Käse korrekt zusammensetzen bzw. das Abschneiden korrekt nachvollziehen. Insbesondere dürfen keine nicht passenden Scheiben angelegt werden.

(6) *Teilaufgabe b bearbeitet*: Bei dieser Aufgabe wird in Teil b explizit nach Erweiterungen gefragt. Ist dieser Teil überhaupt nicht bearbeitet, werden 6 Punkte abgezogen. Werden nur Ideen diskutiert, werden 4 Punkte abgezogen. Eigentlich fragt die Aufgabe nach der Lösung von „Fragestellungen“, so dass die Bearbeitung von zwei Erweiterungen Standard sein sollte. Wird nur eine bearbeitet, diese aber ausführlich und gut, kann auf Punktabzug verzichtet werden. Für besonders interessante, auf eigenen Ideen beruhende Erweiterungen können bis zu 2 Pluspunkte vergeben werden.

(7) *Teil b: Verfahren in Ordnung*: Auch bei den Erweiterungen sollten Korrektheit und angemessene Laufzeiten gegeben sein. Für sehr gute Umsetzungen kann es Pluspunkte, bei Mängeln bis zu 4 Punkten Abzug geben.

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Es muss nachvollziehbar begründet sein, dass nur mögliche Käse gebaut werden.

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

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Es sollten alle vorgegebenen Beispiele bearbeitet und dokumentiert worden sein. Wenn nur etwa die Hälfte dokumentiert ist, gibt es 2 Punkte Abzug.
Es ist in Ordnung, wenn die größeren Beispielausgaben außerhalb der Dokumentation zu finden sind.

(5) *Ergebnisse nachvollziehbar dargestellt*: Es sollten zu jeder gelösten Beispieldatei wenigstens die ersten vier und letzten vier Scheiben angegeben werden. Wichtig ist, dass man einen Eindruck davon bekommen kann, ob das Verfahren korrekt funktioniert.

Aufgabe 3: Pancake Sort

3.1 Lösungsidee

Formalisierung

Ein zu sortierender Pfannkuchenstapel a der Größe n kann als Permutation, dargestellt über ein Tupel $(a_1, a_2, \ldots, a_n)$, verstanden werden, wobei $a_1, \ldots, a_n$ jede ganze Zahl von 1 bis n genau einmal annehmen. Dabei wird davon ausgegangen, dass keine zwei Pfannkuchen die gleiche Größe haben.

Für eine Wende-und-Ess-Operation f_k an der Stelle k wird zunächst ein k-elementiges Präfix der Permutation umgekehrt und das nun erste Element der Permutation (a_k) entfernt.
Im ersten Schritt erhält man

$$b = (a_{k-1}, a_{k-2}, \ldots, a_1, a_{k+1}, a_{k+2}, \ldots, a_n) \tag{3.1}$$

Da a_k entfernt wurde, handelt es sich bei b nun nicht mehr um eine Permutation. Um wieder eine Permutation zu erhalten, werden alle Elemente größer als das entfernte Element a_k um 1 verringert:

$$c_i = \begin{cases} b_i & \text{, falls } b_i \leq a_k \\ b_i - 1 & \text{sonst.} \end{cases}$$

So ergibt sich das Ergebnis der Wende-und-Ess-Operation $f_k(a) = (c_1, \ldots, c_n - 1)$, bei dem es sich auch wieder um eine Permutation handelt.

Ziel ist es nun, mit möglichst wenigen solcher Wende-und-Ess-Operationen die Identitätspermutation $(1, 2, \ldots, n)$ zu erreichen.

Für die Permutation $(2, 3, 1)$ ist dies beispielsweise in zwei Operationen möglich:

$$(2,3,1) \overset{f_2}{\to} (2,1) \overset{f_1}{\to} (1).$$

Mit etwas Ausprobieren lässt sich erkennen, dass es nicht mit weniger als zwei Operationen geht, diese Lösung aber nicht eindeutig ist. In diesem Fall wäre zum Beispiel auch zuerst f_1 und dann f_2 möglich.

Pancake-Sorting im Allgemeinen

Dieses Problem ist eine Abwandlung des *Pancake-Sorting*-Problems, auch *Sorting by prefix reversal* genannt, das 1978 von Gates (ja, *der* Bill Gates) und Papadimitriou bekannt gemacht wurde.[5] Bei diesem Problem wird jeweils nur ein Prefix umgekehrt, jedoch kein Element entfernt. In dieser Form hat das Problem Anwendungen unter anderem in der Bioinformatik. Inzwischen wurde gezeigt, dass das Bestimmen, wie viele reine Wende-Operationen zum Sortieren einer Permutation notwendig sind, in die Klasse der NP-schweren Probleme gehört. Es ist also unwahrscheinlich, dass es einen Algorithmus gibt, der dieses Problem – also ohne Essen nach dem Wenden – in polynomieller Laufzeit lösen kann.

[5] W. H. Gates and C. H. Papadimitriou, “Bounds for sorting by prefix reversal,” Discrete Mathematics, vol. 27, no. 1, pp. 47–57, 1979, doi: 10.1016/0012-365X(79)90068-2.

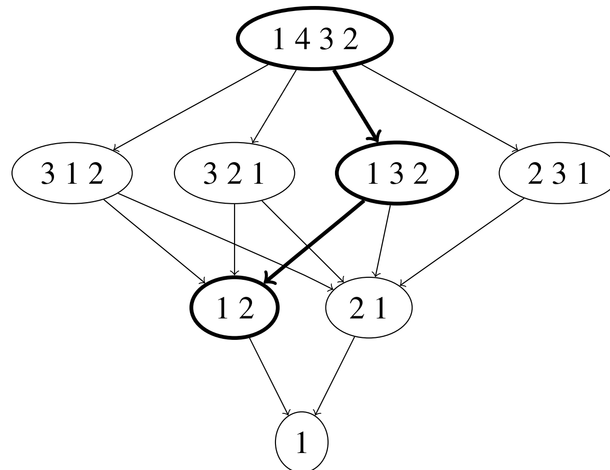


Abbildung 3.1: Der Graph der von $(1,4,3,2)$ erreichbaren Permutationen. Ein möglicher kürzester Weg zu einer Identitätspermutation ist hervorgehoben.

Teilaufgabe a) – A*-Algorithmus

Betrachten wir zunächst nur die Berechnung von $A(s)$, also das Finden einer kürzesten Folge von Wende-und-Ess-Operationen, um eine gegebene Permutation s zu sortieren. Betrachten wir dafür einen gerichteten Graphen $G = (V,E)$, in dem die Knoten jeweils einer Permutation entsprechen und genau dann eine Kante von Permutation a zu b verläuft, wenn die Permutation a in einer Wende-und-Ess-Operation in die Permutation b umgewandelt werden kann. In diesem Graphen suchen wir nun den kürzesten Weg zwischen s und einer Identitätspermutation.

Grafik 3.1 zeigt als Beispiel einen Ausschnitt dieses Graphens mit allen Knoten, die von der Permutation $(1,4,3,2)$ erreichbar sind.

Diesen kürzesten Weg könnte man beispielsweise mit einer Breitensuche, beginnend bei s finden. Durch Verwendung des A*-Algorithmus und einer geeigneten Heuristik kann die Suche jedoch noch maßgeblich beschleunigt werden. Die Heuristik sollte möglichst einfach zu berechnen sein und versuchen, die noch notwendige Anzahl an Operationen zu bestimmen. Damit der Algorithmus korrekt bleibt, sollte sie ebenfalls *zulässig* sein, also grob gesagt die tatsächlich notwendige Anzahl niemals unterschätzen.

Eine Heuristik, die bereits für das ursprüngliche Pancake-Sorting-Problem äußerst gut funktioniert, ist die Breakpoint-Heuristik.[6] Die Idee der Heuristik ist, dass in einer unsortierten Permutation viele „Sprünge“ enthalten sind. Je mehr die Permutation hingegen sortiert wird, desto mehr zusammenhängende Blöcke kommen in ihr vor. Eine Permutation mit weniger Sprüngen könnte also vielversprechender als eine mit mehr Sprüngen sein und sollte zuerst betrachtet werden. Als Breakpoint („Sprung“) wird eine Stelle k in der Permutation a bezeichnet, bei der $|a_k - a_{k+1}| \neq 1$, also zwei benachbarte Positionen keine benachbarten Zahlen enthalten. Zusätzlich kann auch an der Stelle $n+1$ ein Breakpoint sein – genau dann, wenn $a_n \neq n$ ist. Die Permutation $(1,2,4,3)$ hat beispielsweise Breakpoints an den Stellen 2 und 4. Sei $g(a)$ die Anzahl solcher Breakpoints.

Es kann gezeigt werden, dass eine Wende-und-Ess-Operation maximal 3 Breakpoints entfernen kann, also $g(f_k(a)) \geq g(a) - 3$ gilt.

[6] M. Helmert, “Landmark Heuristics for the Pancake Problem,” SOCS, vol. 1, no. 1, pp. 109–110, Aug. 2010, doi: 10.1609/socs.v1i1.18176.

Algorithmus 6 A*-Algorithmus zum Lösen einer Permutation s

```
procedure ASTERN(s)
    closed ← ∅                                          ▷ Liste der bearbeiteten Knoten
    open ← {(s,0)}
    while open ≠ ∅ do
                        ▷ Minimum nach Summe aus bisheriger Distanz und Heuristik
        a ← open.popMinimum()
        if a ist sortiert then
            return a
        end if
        if a ∈ closed then
            continue
        else
            closed.insert(a)
        end if
        for 1 ≤ i ≤ a.length do
                                ▷ s.length − a.length ist der Abstand zum Startknoten
            open.insert((f_i(a), s.length − a.length + h(f_i(a))))
        end for
    end while
end procedure
```

Beweis: Durch die Wende-Operation kann maximal ein Breakpoint entfernt werden, da sich nur ein Paar benachbarter Elemente verändert, nähmlich zwischen a_k und a_{k+1}. Durch das Entfernen des ersten Elements kann ebenfalls maximal ein Breakpoint entfernt werden, nähmlich zwischen b_1 und b_2. Durch die Anpassung der Zahlen größer als a_k kann ebenfalls maximal ein Breakpoint entfernt werden: Sollte an einer beliebigen Stelle i vorher $a_i = a_k - 1$ und $a_{i+1} = a_k + 1$ oder $a_i = a_k + 1$ und $a_{i+1} = a_k - 1$ zugetroffen haben, so verschwindet durch die Anpassung ein Breakpoint. Da jede Zahl nur maximal einmal in einer Permutation vorkommt, kann es nur eine solche Position geben. Wenn in jedem Schritt maximal 3 Breakpoints entfernt werden, kann eine Permutation niemals mit weniger als $h(a) = g(a)/3$ Schritten gelöst werden. Die Heuristik $h(a)$ unterschätzt also niemals die tatsächlich noch notwendige Anzahl an Schritten und ist eine zulässige Heuristik. Die meisten Permutationen benötigen mehr als $g(a)/3$, da die Suche jedoch sehr schnell sehr breit wird reicht auch eine deutlich überschätzende Heuristik aus, um die Suche deutlich zu beschleunigen.

Der A*-Algorithmus basiert dann im Grundprinzip auf einer Breitensuche. Die Reihenfolge, in der die Knoten bearbeitet werden, wird aber verändert. Es werden zuerst solche Knoten bearbeitet, bei denen die Summe aus der bisherigen Distanz und der mindestens noch zu erwartenden Distanz nach der Heuristik möglichst niedrig ist. So werden die Knoten zuerst bearbeitet, die eine voraussichtlich kürzere Gesamtlösung erreichen, und der Algorithmus kommt schneller zum Ziel.

So ergibt sich Algorithmus 6, wobei zur Rekonstruktion der tatsächlichen Lösung bei der Implementation natürlich noch der jeweilige Vorgänger für jeden besuchten Knoten gespeichert werden muss.

Laufzeit

Die Anzahl der erreichbaren Permutationen der Größe $|s| \geq \frac{n}{2}$ ist maximal

$$n \cdot (n-1) \cdot \ldots \cdot \frac{n}{2}.$$

Die Anzahl aller Permutationen der Größe $|s| \leq \frac{n}{2}$ ist

$$\frac{n}{2} \cdot (\frac{n}{2} - 1) \cdot \ldots \cdot 1,$$

denn hier greift die Anpassung der Permutation nach einer Wende-und-Ess-Operation. Insgesamt lassen sich beide Hälften nach oben durch $\mathcal{O}(n^{\frac{n}{2}})$ abschätzen.

Die Gesamtlaufzeit hängt nun vom genauen Algorithmus ab. Bei einer naiven Breitensuche werden wir eine Laufzeit von $\mathcal{O}(n^{\frac{n}{2}+2})$ haben, da bei jeder Permutation alle $\mathcal{O}(n)$ Nachfolgepermutationen betrachtet werden müssen und jede in $\mathcal{O}(n)$ normalisiert werden muss. Der A*-Algorithmus hat dahingegen $|V| = \mathcal{O}(n^{\frac{n}{2}})$ und entsprechend eine Worst-Case Laufzeit von $\mathcal{O}(n^{\frac{n}{2}} \cdot n^2 \cdot \log n)$, da jeder Durchlauf der Schleife eine Laufzeit von

$$\mathcal{O}(\log|V| + n \cdot (n + \log|V|)) = \mathcal{O}(n^2 \cdot \log n)$$

benötigt. Um die Laufzeit zu verbessern, könnte man zum Beispiel einen Fibonacci-Heap oder Hollow-Heap verwenden. Damit würde sich die Gesamtlaufzeit zu $\mathcal{O}(n^{\frac{n}{2}} \cdot n^2)$ verbessern.

Die Speicherkomplexität ist dabei insgesamt $\mathcal{O}(n \cdot n!)$.

Teilaufgabe b) – Dynamische Programmierung

Um Teilaufgabe b) zu lösen, könnte der bereits beschriebene A*-Algorithmus für jede Permutation einer bestimmten Länge aufgerufen werden, um alle $A(s)$ und daraufhin $P(n)$ zu bestimmen. Dabei würde jedoch nicht ausgenutzt, dass die Ergebnisse einer bereits untersuchten Permutation beim Lösen einer anderen hilfreich sein könnten. Daher lässt sich $P(n)$ deutlich effizienter über dynamische Programmierung lösen.

Um dynamische Programmierung anwenden zu können, muss eine Instanz eines Problems (in diesem Fall eine Permutation) sich optimal lösen lassen, indem zunächst kleinere Teilprobleme gelöst und aus den Ergebnissen dann die optimale Lösung bestimmt werden kann (Optimalitätsprinzip bzw. *optimal substructure*). Ferner sollten die Teilprobleme jeweils für mehrere größere Probleme benötigt werden, damit sich die dynamische Programmierung tatsächlich lohnt (*overlapping subproblems*). Die SOLVE-Prozedur in Algorithmus 7 berechnet die minimale Länge, in der eine Permutation sortiert werden kann, indem sie eine Operation anwendet und die dann entstandenen kleineren Teilprobleme rekursiv löst – die beiden Bedingungen treffen also zu.

Indem die Zwischenergebnisse der rekursiven Funktion z. B. in einer Hash-Tabelle gespeichert werden und gegebenenfalls wiederverwendet werden, kann vermieden werden, dass gleiche Teilprobleme mehrmals berechnet werden. Wenn $A(n)$ bestimmt werden soll, kann die Tabelle auch zwischen den Berechnungen von $P(s)$ beibehalten werden und so können auch vorherige Teilergebnisse wiederverwendet werden. So ergibt sich Algorithmus 7, um eine einzelne Permutation sowie $P(n)$ zu berechnen. Um die gefundenen Lösungen zu rekonstruieren müsste

Algorithmus 7 Rekursives Lösen

```
procedure SOLVE(a,dp)
    if a ∈ dp then
        return dp[a]
    end if
    if a ist sortiert then
        return 0
    end if
    x ← ∅
    for 1 ≤ i ≤ a.length do
        x.insert(SOLVE(f_i(a)))
    end for
    dp[a] ← min(x)
    return min(x)
end procedure

procedure BERECHNEP(n)
    dp ← ∅
    P ← 0
    for a Permutation der Länge n do
        P ← min(P, SOLVE(s, dp))
    end for
    return P
end procedure
```

natürlich nicht nur die Länge der besten Sortierung sondern auch die dafür notwendigen Schritte zurückgegeben und gespeichert werden.

Bei einigen anderen (vermeintlich effizienteren) Ansätzen mit dynamischer Programmierung ist Vorsicht geboten. Ein denkbarer Ansatz könnte ein Problem lösen, indem er eine optimale Lösung für die ersten $n-1$ Stellen sucht und daraufhin entscheidet, ob an der Stelle n eine Wende-und-Ess-Operation stattfinden soll oder nicht. Dieser Algorithmus würde zwar bei vielen Beispielen gut funktionieren, jedoch nicht bei allen. Er betrachtet nur Lösungen, bei denen der Pfannenwender zuerst auf höhere Pfannkuchen und dann auf niedrigere angewendet wird. Im Allgemeinen reicht es nicht, nur zu bestimmen, an welchen Stellen der Pfannenwender angewendet werden soll – auch die Reihenfolge der Operationen ist wichtig. Ein korrekter Algorithmus muss auch Lösungen betrachten, bei denen z. B. zuerst beim 5., dann beim 1. und zuletzt beim 2. Pfannkuchen gewendet und gegessen wird.

Für $P(n)$, die mit dem garantiert optimalen Algorithmus nicht mehr lösbar sind, könnten auch untere Schranken von Interesse sein. Dann kann das Problem als Optimierungsproblem (Finde einen Pancake-Stapel mit möglichst großem $A(s)$.) verstanden werden und Optimierungsverfahren wie z. B. genetische Algorithmen angewendet werden.

Laufzeit

Die Prozedur SOLVE wird während des gesamten Ablaufs nur maximal einmal für jedes Mögliche Teilproblem aufgerufen. Pro Aufruf benötigt die SOLVE -Prozedur maximal eine Laufzeit

von $\mathcal{O}(n^2)$ (angenommen, dass die Suche in der Hash-Tabelle in $\mathcal{O}(1)$ abläuft). Um $P(n)$ zu bestimmen, müssen maximal alle Permutationen mit Länge n oder weniger gelöst werden, also $\mathcal{O}(n!)$, wie bereits beim A*-Algorithmus gezeigt. So ergibt sich für die Gesamtlaufzeit

$$T(n) \in \mathcal{O}(n^2) \cdot |L| = \mathcal{O}(n^2 \cdot n!).$$

Der Speicherbedarf pro Teilproblem wächst linear mit der Größe der Permutation, da lediglich die Permutation und eine weitere Zahl gespeichert wird. So ergibt sich eine Speicherkomplexität von $S(n) \in \mathcal{O}(n! \cdot n)$. Die beiden Algorithmen haben also die selbe theoretische Laufzeit und auch dieser Algorithmus ist nur für relativ kleine Beispiele bzw. n geeignet. Da bei dynamischer Programmierung größerer Speicherbedarf für eine kürzere Laufzeit in Kauf genommen wird, hat dieser Ansatz inhärent einen relativ großen Speicherplatzbedarf. Dennoch lassen sich damit $P(n)$ bis $n = 12$ in wenigen Minuten berechnen.

Erweiterungen

Eine bekannte Variante von Pancake Sort ist das „Burnt Pancake Problem“, bei dem die Pfannkuchen eine Orientierung haben (bei jedem wird eine Seite als „angebrannt“ markiert) und am Ende der Sortierung alle Pfannkuchen die gleiche Orientierung haben müssen (alle angebrannten Seiten liegen unten). Auch hierfür kann man versuchen, $A(S)$ und $P(n)$ zu bestimmen.

3.2 Beispiele

Im Folgenden sind einige optimale Lösungen (also kürzeste sortierende Folgen von Operationen) für die Beispieleingaben angegeben. Es kann jedoch auch andere Lösungen der gleichen Länge geben. Die Zahl vor dem Pfeil gibt jeweils an, wie viele Pfannkuchen gedreht werden. Mit dem A*-Algorithmus können alle Beispieleingaben in meist deutlich unter 100 ms gelöst werden. Zufällige Permutationen sind bis zu einer Länge von über 25 Pfannkuchen so noch lösbar.

Zunächst aber eine Übersicht über die Zahlen $A(S)$ für die Beispieleingaben:

`pancake*.txt`	0	1	2	3	4	5	6	7
$A(S)$	2	3	4	6	7	6	8	8

pancake0.txt

```
(3, 2, 4, 5, 1)
5 -> (5, 4, 2, 3)
4 -> (2, 4, 5)
```

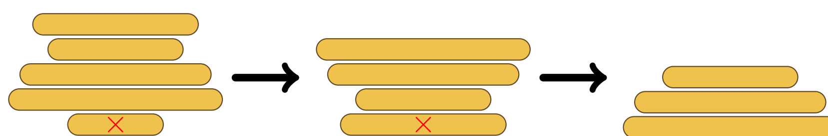


Abbildung 3.2: Lösung für Beispiel 0

pancake1.txt

```
(6, 3, 1, 7, 4, 2, 5)
7 -> (2, 4, 7, 1, 3, 6)
3 -> (4, 2, 1, 3, 6)
4 -> (1, 2, 4, 6)
```

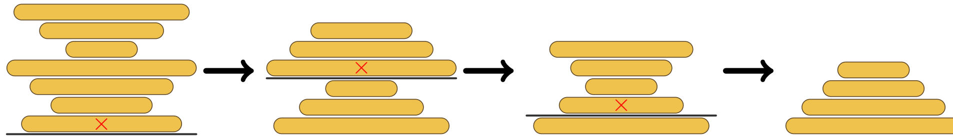


Abbildung 3.3: Lösung für Beispiel 1

pancake2.txt

```
(8, 1, 7, 5, 3, 6, 4, 2)
4 -> (7, 1, 8, 3, 6, 4, 2)
4 -> (8, 1, 7, 6, 4, 2)
2 -> (8, 7, 6, 4, 2)
5 -> (4, 6, 7, 8)
```

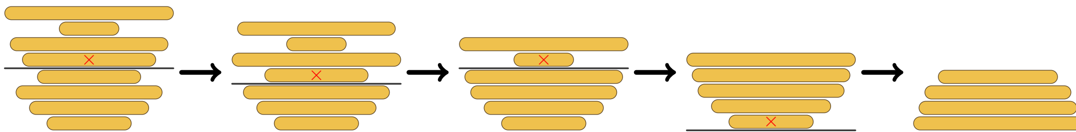


Abbildung 3.4: Lösung für Beispiel 2

pancake3.txt

```
( 5, 10,  1, 11,  4,  8,  2,  9,  7,  3,  6)
10 -> ( 7,  9,  2,  8,  4, 11,  1, 10,  5,  6)
 8 -> ( 1, 11,  4,  8,  2,  9,  7,  5,  6)
 4 -> ( 4, 11,  1,  2,  9,  7,  5,  6)
 6 -> ( 9,  2,  1, 11,  4,  5,  6)
 1 -> ( 2,  1, 11,  4,  5,  6)
 3 -> ( 1,  2,  4,  5,  6)
```

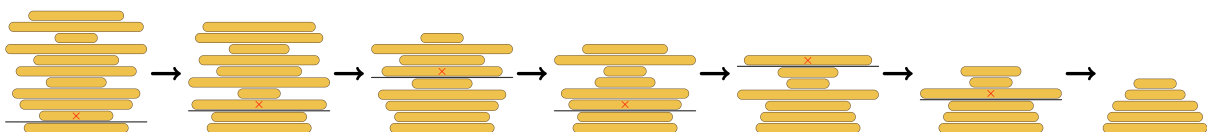


Abbildung 3.5: Lösung für Beispiel 3

pancake4.txt

```
( 7,  4, 11,  5, 10,  6,  1, 13, 12,  9,  3,  8,  2)
11 -> ( 9, 12, 13,  1,  6, 10,  5, 11,  4,  7,  8,  2)
 5 -> ( 1, 13, 12,  9, 10,  5, 11,  4,  7,  8,  2)
 7 -> ( 5, 10,  9, 12, 13,  1,  4,  7,  8,  2)
 6 -> (13, 12,  9, 10,  5,  4,  7,  8,  2)
 9 -> ( 8,  7,  4,  5, 10,  9, 12, 13)
 5 -> ( 5,  4,  7,  8,  9, 12, 13)
 1 -> ( 4,  7,  8,  9, 12, 13)
```

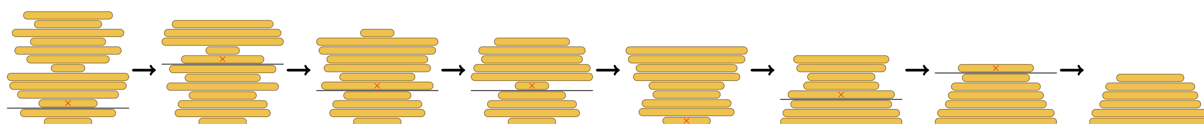


Abbildung 3.6: Lösung für Beispiel 4

pancake5.txt

```
( 4, 13, 10,  8,  2,  3,  7,  9, 14,  1, 12,  6,  5, 11)
 1 -> (13, 10,  8,  2,  3,  7,  9, 14,  1, 12,  6,  5, 11)
13 -> ( 5,  6, 12,  1, 14,  9,  7,  3,  2,  8, 10, 13)
 7 -> ( 9, 14,  1, 12,  6,  5,  3,  2,  8, 10, 13)
 9 -> ( 2,  3,  5,  6, 12,  1, 14,  9, 10, 13)
 5 -> ( 6,  5,  3,  2,  1, 14,  9, 10, 13)
 6 -> ( 1,  2,  3,  5,  6,  9, 10, 13)
```

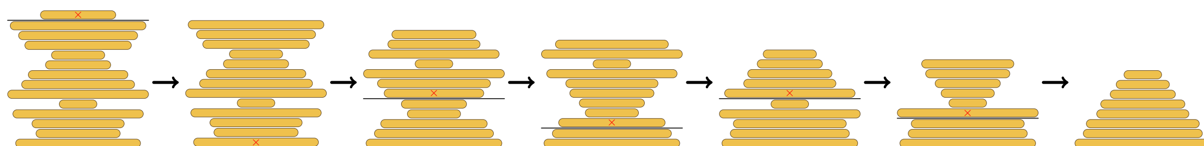


Abbildung 3.7: Lösung für Beispiel 5

pancake6.txt

```
(14,  8,  4, 12, 13,  2,  1, 15,  7, 11,  3,  9,  5, 10,  6)
15 -> (10,  5,  9,  3, 11,  7, 15,  1,  2, 13, 12,  4,  8, 14)
 7 -> ( 7, 11,  3,  9,  5, 10,  1,  2, 13, 12,  4,  8, 14)
10 -> (13,  2,  1, 10,  5,  9,  3, 11,  7,  4,  8, 14)
 9 -> (11,  3,  9,  5, 10,  1,  2, 13,  4,  8, 14)
 2 -> (11,  9,  5, 10,  1,  2, 13,  4,  8, 14)
 9 -> ( 4, 13,  2,  1, 10,  5,  9, 11, 14)
 2 -> ( 4,  2,  1, 10,  5,  9, 11, 14)
 4 -> ( 1,  2,  4,  5,  9, 11, 14)
```

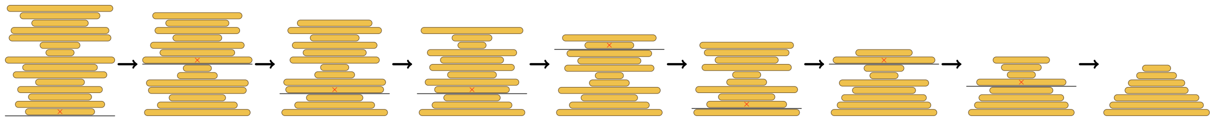


Abbildung 3.8: Lösung für Beispiel 6

pancake7.txt

```
( 8,  5, 10, 15,  3,  7, 13,  6,  2,  4, 12,  9,  1, 14, 16, 11)
11 -> ( 4,  2,  6, 13,  7,  3, 15, 10,  5,  8,  9,  1, 14, 16, 11)
 8 -> (15,  3,  7, 13,  6,  2,  4,  5,  8,  9,  1, 14, 16, 11)
 1 -> ( 3,  7, 13,  6,  2,  4,  5,  8,  9,  1, 14, 16, 11)
 2 -> ( 3, 13,  6,  2,  4,  5,  8,  9,  1, 14, 16, 11)
12 -> (16, 14,  1,  9,  8,  5,  4,  2,  6, 13,  3)
11 -> (13,  6,  2,  4,  5,  8,  9,  1, 14, 16)
 8 -> ( 9,  8,  5,  4,  2,  6, 13, 14, 16)
 6 -> ( 2,  4,  5,  8,  9, 13, 14, 16)
```

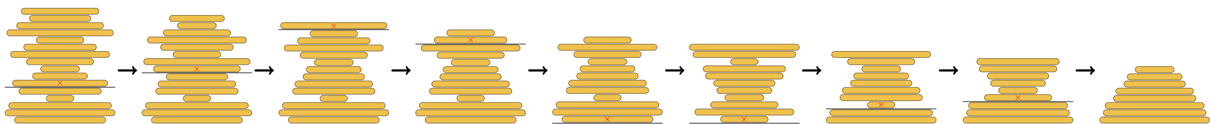


Abbildung 3.9: Lösung für Beispiel 7

Teilaufgabe b)

Tabelle 2 zeigt die berechneten $P(n)$-Werte. Bei $P(14)$ und $P(15)$ handelt es sich lediglich um untere Schranken für $P(n)$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(n)$	0	1	2	2	3	3	4	5	5	6	6	7	7	≥ 8	≥ 9

Tabelle 2

Tabelle 3 zeigt für jedes n beispielhaft eine Permutation, die $A(s) = P(n)$ hat, also für ihre Länge maximal viele Wende-und-Ess-Operationen zum Sortieren benötigt.

n	Permutation s mit $A(s) = P(n)$
2	(2, 1)
3	(2, 3, 1)
4	(4, 2, 3, 1)
5	(4, 2, 5, 3, 1)
6	(6, 5, 2, 4, 3, 1)
7	(7, 3, 6, 2, 5, 4, 1)
8	(4, 8, 2, 6, 3, 7, 5, 1)
9	(9, 6, 3, 8, 5, 2, 7, 4, 1)
10	(9, 1, 7, 4, 10, 6, 3, 8, 5, 2)
11	(11, 10, 3, 5, 8, 2, 7, 1, 9, 4, 6)
12	(12, 1, 10, 2, 6, 11, 4, 7, 9, 5, 8, 3)
13	(13, 9, 6, 10, 2, 5, 11, 7, 3, 8, 1, 12, 4)
14	(4, 12, 6, 3, 11, 8, 1, 10, 7, 2, 13, 9, 14, 5)
15	(1, 9, 2, 10, 3, 11, 4, 12, 5, 13, 6, 14, 7, 15, 8)

Tabelle 3

3.3 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Die Modellierung darf keine Hürde für die Implementierung sein. Für die Modellierung der Stapel bieten sich Permutationen an.

(2) *Verfahren nicht unnötig ineffizient*: Werden bei einer DP-Lösung beispielsweise nach der Wende-Und-Ess-Operation die Zahlen nicht so angepasst, dass wieder eine Permutation entsteht, so vergrößert dies die Gesamtzahl der Teilprobleme signifikant und wird mit 2 Punkten Abzug bewertet.

(3) *Teil a: Laufzeit des Verfahrens in Ordnung*: Der Algorithmus sollte in der Lage sein, alle BWINF-Beispieleingaben zu lösen. Dafür darf die theoretische Laufzeitkomplexität nicht schlechter als $\mathcal{O}(n!)$ sein.

(4) *Teil b: Laufzeit des Verfahrens in Ordnung*: Sollte Teil b komplett fehlen, werden 4 Punkte abgezogen. Wird bei Teil b nur der Algorithmus aus Teil a für alle Permutationen verwendet und keine weitere Idee, um die Laufzeit zu verringern, werden 2 Punkte abgezogen. Abzug gibt es auch, wenn die Werte $P(n)$ nicht bis einschließlich $P(11)$ berechnet werden können. Können alle Werte bis $P(13)$ berechnet werden, gibt es 2 Pluspunkte.

(5) *Speicherbedarf in Ordnung*: Der Speicherbedarf sollte nicht so groß sein, dass die Beispieleingaben bzw. Teilaufgabe b) dadurch nicht mehr lösbar sind, also auf keinen Fall mehr als $\mathcal{O}(n!)$ betragen.

(6) *Verfahren mit korrekten Ergebnissen*: Die bei Teil a berechneten Operations-Folgen müssen gültig sein (also den Stapel sortieren) und die optimale Länge haben. Bei Teil b müssen die berechneten Werte korrekt sein.
Sowohl auf- als auch absteigend sortierte Pfannkuchenstapel werden akzeptiert.

(7) *Teilaufgabe b bearbeitet*: Bei dieser Aufgabe ist in Teil b die Berechnung der PWUE-Zahlen $P(n)$ gefragt. Ist dieser Teil überhaupt nicht bearbeitet, werden 6 Punkte abgezogen. Sind nur Ideen diskutiert, werden 4 Punkte abgezogen.

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Das Verfahren und insbesondere seine Korrektheit müssen gut nachvollziehbar begründet werden. Insbesondere bei DP-Ansätzen sollte zumindest ansatzweise begründet sein, dass das Verfahren immer die optimale Lösung findet.
Für jede Optimierung muss erklärt werden, warum diese eine (deutliche) Effizienzsteigerung bringt, gerade wenn die Komplexitätsklasse unverändert bleibt.

(2) *NP-Schwere des Problems erkannt*: Falls die NP-Schwere erkannt und korrekt begründet wird, etwa durch eine (informelle) Reduktion von einem bekannten NP-schweren Problem, können Pluspunkte vergeben werden.

(3) *Gute Überlegungen zur Komplexität des Verfahrens*: Die Laufzeit- und Speicherkomplexität sollte korrekt bestimmt werden. Ist die Analyse besonders formal oder genau, können dafür Pluspunkte vergeben werden.

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Für Teil a sollen für alle BWINF-Beispiele Ergebnisse angegeben sein. Wenn nur etwa die Hälfte dokumentiert ist, gibt es 2 Punkte Abzug.

(4) *PWUE-Zahlen dokumentiert*: (Teil b) Für $n = 8, \ldots, 11$ müssen die Werte $P(n)$ angegeben werden sowie Beispielstapel, für die tatsächlich $P(n)$ Wendeoperationen nötig sind. Fehlen die Beispiele, wird 1 Punkt abgezogen. Können durch theoretische Überlegungen $P(n)$ für $n > 13$ angegeben werden, gibt es Pluspunkte.

(6) *Ergebnisse nachvollziehbar dargestellt*: Alle Wende-und-Ess-Operationen sollten nachvollziehbar angegeben sein. Mindestens eine gefundene optimale Lösung pro Beispiel muss mit allen Zwischenschritten dargestellt werden. Sinnvoll sind auch die Angabe des sortierten Stapels und die Angabe der (optimalen) Anzahl der benötigten Operationen; ein Fehlen führt aber nicht zu Punktabzug. Eine grafische Darstellung ist nicht gefordert, kann aber, wenn sie inhaltlich sinnvoll ist, positiv bewertet werden.

Aus den Einsendungen: Perlen der Informatik

Aufgabe 1:Weniger krumme Touren

Es gibt keine Route, die der Anforderung an die Abbiegwinkel genüge tut.

Sie brauchen die nicht Python inklusive Bücherei stopit für mein Programm.

Aufgabe 2: Alles Käse

```
exit() # Vorbei bye bye
```

den lowestlargest Wert

Für die Speicherung wird ein Käseblock-Objekt verwendet [...] das die Käsescheiben speichert.

Daher kann, wenn man mehrere Scheiben hat, die an den Quader angepuzzelt werden können, die aber nicht identisch sind (falls sie das wären wäre es egal), nur die Scheibe zur Vervollständigung führen, deren Kantenlänge, welche die Käsescheiben nicht gemein haben, kleiner ist als die der anderen Käsescheibe(n), da sie, falls man eine andere anpuzzeln würde, nicht mehr angepuzzelt werden kann, da die Kantenlänge, die sie nicht gemein hat, nun nicht mehr am Quader zu finden ist, da die Kantenlänge des Quaders, die vorher diese Länge hatte nun um 1 erhöht ist, da der größere Quader angepuzzelt wurde und sich damit die Kantenlänge erhöht hat, die dieser nicht hatte (s. o.), und die andere Kantenlänge, die sie nicht mit der anderen Käsescheibe gemein hatte größer war und falls die gemeinsame Kantenlänge kleiner oder gleich dieser Kantenlänge ist würde sie aus den selben Gründen keine Kantenlänge finden, die ihr kleiner oder gleich ist, wenn man die nicht gemeinsame nun an ihre Stelle legen würde, und die Kanten nicht mehr kleiner werden, da man nur Scheiben anpuzzelt.

Aufgabe 3: Pancake Sort

Beachte aber, dass du dann möglicherweise nicht mehr zu Lebzeiten eine Ausgabe bekommst!

Das bedeutet, es ist nicht sicher, dass dies die echten PWUE-Zahlen sind. (Ich habe das Programm aber natürlich mehrmals ausprobiert, weswegen ich mir trotzdem sicher genug bin, um die Ergebnisse hier aufzuführen.)

Der Lösungsansatz benutzt eine Schleife, um die Zahlen in der Sequenz zu sortieren.