29. Bundeswettbewerb Informatik 2010/2011
2. Runde

Tim Taubner, Verwaltungsnummer 29.108.01

7. April 2011

Dies ist die Dokumentation zu den von mir bearbeiteten Aufgaben 1 und 2 der 2. Runde des 29. Bundeswettbewerbs Informatik 2010/2011. Die mir zugeteilte Verwaltungsnummer ist 29.0108.01. Für alle Aufgabe werden jeweils die Lösungsidee und eine Programm-Dokumentation angegeben, sowie geeeignete Programm-Ablaufprotokolle und der Programm-Text selbst. Auf die ausführbaren Lösungen wird in der Dokumentation verwiesen. Der Quelltext ist beigefügt. Ebenfalls enthalten sind weiterführende Gedankengegänge, diese erhalten ebenfalls einen eigenen Unterpunkt. In diesem ist sowohl kurz die Idee als auch die Implementationerläuterung enthalten. Zusätzlich ist am Ende eine allgemeine Beschreibung enthalten, wie die erstellten Programme von der mitgelieferten CD aus gestartet werden können. Alle eingereichten Quelldateien, Kunsterzeugnisse (wie z.B. Bilder) und ausführbare Programmdistributionen wurden alleine von mir, Tim Taubner, erstellt.

Inhaltsverzeichnis

A. Allgemeines

1. Persönliche Anmerkungen

Der BWInf und ich Der Bundeswettbewerb Informatik konnte mich sehr begeistern. Viel konnte ich bereits durch die 1. Runde lernen, z.B. wie eine gute Dokumentation erstellt werden kann. Auch das Ergebnis lässt sich sehen. Wenn ich gefragt werde was ich eigentlich am PC mache, klappe ich mein Laptop auf und zeige die Dokumentation zur 1. Aufgabe[1]. Viel besser kann man finde ich nicht zeigen, dass Informatik *nicht* nur Programmieren ist.

[1] siehe Einsendung zur 1. Runde, Einsendungsnr. 108, besonders Aufgabe 1

Wahl der Programmiersprache Die benutzte Programmiersprache ist durchgehend Scala. Das liegt einfach an meiner Neigung, kurzen und dichtgepackten[2] Code zu schreiben. Ich bin mir durchaus bewusst, dass Scala - noch - keine weit verbreitete Sprache ist. Aber ich denke, dass es auch einem Scala-fremden Informatiker gefällt, wenn aussagekräftiger Code abgegeben wird.

Dank Ich erlaube mir hier, Personen zu danken, die mir zu dieser Einsendung verholfen haben. Auch wenn alle Leistungen im Sinne des Wettbewerbs von mir erbracht wurden, habe ich dazu nicht wenig Energie aus der Umgebung gezogen.[3] Zum einen meiner Freundin[4], aber auch meiner Familie. Sie scheinen bereits ein Algorithmus entwickelt haben, mit meinen einsilbigen Antworten fertig zuwerden.

2. Dateistruktur der CD

Die Dateien auf der CD sind folgendermaßen strukturiert. Jede Aufgabe hat einen Ordner AufgabeX mit den beiden folgenden Unterordnern.

src	Unterordner, in dem die Quelltexte in der Paketstruktur (de/voodle/..) liegen
dist	Unterordner, in dem ausführbare Dateien oder - wie bei Aufgabe 1 - andere Erzeugnisse sowie benötigte Bibliotheken enthalten sind

Zusätzlich ist die vorliegende Dokumentation digital unter **TeX-Doku-Einsendung-2910801-Tim-Taubner.pdf** im Wurzelverzeichnis zu finden. Auch die LaTeX-Quelldateien, mit der diese Dokumentation erzeugt wurden, ist im Verzeichnis "TeX-Doku-Quelldateien" zu finden.

3. Ausführvoraussetzungen

Um die Programmbeispiele ausführen zu können, müssen folgende Voraussetzungen erfüllt werden:

Java Runtime: Mindestens Version 1.5 (Java 5), empfohlen: ≥ 1.6

Prozessor: Mindestens 1 GHz, empfohlen: ≥ 1.6 GHz

Arbeitsspeicher: Mindestens 512 MB, empfohlen: ≥ 1 GB

Grafikkarte: beliebig

Getestete Betriebsysteme: Windows 7, Windows XP, Linux (Ubuntu 10.04, Kubuntu 10.10)

4. Starten der Programme

Wurzelverzeichnis Im Wurzelverzeichnis der CD beginnt die Ordnerhierarchie der mitgelieferten Dateien. Stellen Sie bitte sicher, dass Sie im Wurzelverzeichnis sind, bevor Sie die in den jeweiligen Aufgaben beschriebene Startanleitungen ausführen. (z.B. durch neues Starten der Kommandozeile gemäß folgender Anleitung)

[2] Hier zeigt sich eine Schwäche der deutschen Sprache: Sie ist *verbose*. Ich versuche hier *concise* aus dem Englischem zu übersetzen

[3] Vergleichen Sie dies mit einem Eisberg, er zieht beim schmilzen die ganze Wärme aus der Umgebung.

[4] Meine eigene Perle der Informatik ;-]

Starten der Kommandozeile Da die meisten mitgelieferten Programme aus der Kommandozeile gestartet werden müssen, soll hier kurz erläutert werden, wie Sie die Kommandozeile unter den gängigeren Betriebsystemen starten können. [5]

Unter Windows Unter Windows starten Sie die Kommandozeile durch: Start → Ausführen → 'cmd' eingeben → Kommandozeile. Nun können Sie durch Angabe des Laufwerkbuchstabens des CD-Laufwerks (z.B. "E:") auf das Wurzelverzeichnis der CD wechseln. Alle Befehle können nun durch Copy&Paste entsprechend der Nutzungsdokumentation der jeweiligen Aufgabe ausgeführt werden. Z.B. für Aufgabe 3 mit:

java -jar Aufgabe1/target/Kisten.jar

Unter GNOME Unter gängigeren GNOME Distributionen wie z.B. Ubuntu 8 starten sie die Kommandozeile durch: Applikationen → System → Terminal. Die CD wird unter Standarddistributionen unter /media/disk o.ä. eingehängt. Wechseln Sie durch 'cd /media/disk' in das Wurzelverzeichnis der CD. Alle Befehle können nun durch Copy&Paste entsprechend der Nutzungsdokumentation der jeweiligen Aufgabe ausgeführt werden. Z.B. für Aufgabe 3 mit:

java -jar Aufgabe1/target/Kisten.jar

Unter KDE Unter gängigeren KDE-Distributionen wie z.B. Kubuntu 9 starten sie die Kommandozeile durch: Start → Applikationen → System → Terminal. Die CD wird unter Standarddistributionen unter /media/disk o.ä. eingehängt. Wechseln Sie durch 'cd /media/disk' in das Wurzelverzeichnis der CD. Alle Befehle können nun durch Copy&Paste entsprechend der Nutzungsdokumentation der jeweiligen Aufgabe ausgeführt werden. Z.B. für Aufgabe 3 mit:

java -jar Aufgabe1/target/Kisten.jar

Unter Mac OS X Leider steht mir kein Mac zur Benutzung bereit, das Öffnen der Konsole sollte jedoch entweder selbsterklärend oder ähnlich der unter KDE/GNOME sein. (Beachten Sie bitte, dass Aufgabe2 leider nicht unter Mac OS X lauffähig ist)

[5] Ich respektiere Ihre wahrscheinlich umfassenden Kenntnisse mit Perl. Die Anleitung zum Kommandozeilestart dient lediglich der Vollständigkeit.

B. Erste bearbeitete Aufgabe: (1) Kisten in Kisten in Kisten

1. Lösungsidee

1.1. Allgemein

Ein *Kistenbaum* ist ein *binärer Baum* von Kisten, indem alle Knoten gleichzeitig in ihren gemeinsamen Vorgänger passen.
Als einen *Kistensatz* bezeichne ich eine Menge von Wurzeln mehrerer Kistenbäumen.
Mit $elems(ks)$ sei die Vereinigung der Menge aller Kisten aus den Kistenbäumen bezeichnet.
Das Grundproblem ist nun, zu einer Menge gegebener Kisten $k_1, k_2, \ldots, k_n$ ein Kistensatz ks mit den Wurzeln $w_1, \ldots, w_m$ zu erzeugen mit $elems(ks) = \{k_1, k_2, \ldots, k_n\}$. Seien $v_1, \ldots, v_m$ die jeweiligen Volumina der Wurzeln $w_1, \ldots, w_m$. Dann gilt es als weitere Aufgabe $\sum_{i=0}^{m} v_i$ zu minimieren.

1.2. Bruteforce

Die Liste wird entsprechend dem Volumen von groß nach klein sortiert. Die Liste wird nacheinander zu Kartonsätzen kombiniert. Eine Hilfsfunktion erzeugt aus einer Menge von Kartonsätzen durch hinzufügen einer gegebenen Kiste die Menge aller möglichen Kistensätze. Diese werden dann weiter mit dem nächsten zu noch mehr Kistensätzen kombiniert. Am Ende sind alle Elemente der Liste abgearbeitet.
Im Folgenden ist der Algorithmus in Scala Code dargestellt. Ich habe mich bewusst gegen Pseudo-Code Notation entschieden. Der Scala Code ist meiner Meinung nach ebenso effektiv wie Pseudo-Code. Lediglich wenige Elemente funktionaler und objektorientierter Elemente müssen dem Leser bekannt sein.[6]
Wichtig ist, um den Code zu verstehen, dass `(satz ++< kiste)` alle Möglichkeiten erzeugt, wie man die Kiste in einen Satz einfügen kann. [7]

```
// Nacheinander die Kisten "auffalten" mit Hilfe der hilfsPacken Funktion
def packe = (Set[KistenSatz]() /: kisten) ( hilfsPacken )
def hilfsPacken(sätze: Set[KistenSatz], kiste: Kiste) =
  if(sätze.isEmpty)
    Set(KistenSatz(kiste :: Nil)) // KistenSatz nur mit der Kiste
  else
    (Set[KistenSatz]() /: sätze) { // Beginne mit leerer Menge
      (menge, satz) =>
       menge ++  // Füge neue Möglichkeiten der menge hinzu
       (satz ++< kiste) // Erzeugt neue Möglichkeiten
    }
```

Laufzeitverhalten Das Laufzeitverhalten dieses Algorithmus ist fatal. Es muss im letzten Schritt eine Kiste in bis zu $(n-1)!$ Kistensätze gepackt werden. Die Laufzeit eine Kiste in einen Kistensatz zu packen ist $O(n)$, es muss für jede Kiste des Kistensatzes Möglichkeiten erzeugt werden. Wir erhalten also $n! + n \cdot (n-1)! + (n-1) \cdot (n-2)! + \cdots + 2 \cdot 1! + 1$. Sprich $O(n \cdot n!)$.
Auch wenn der worst-case meist nicht erreicht wird, beispielsweise wenn es für 40 Kisten eher 20! Möglichkeiten gibt, würde die Berechnung aller Möglichkeiten bereits $8 \cdot 10^{15}$ Jahre

[6] Beispielsweise sollten Sie wissen, wie foldLeft, currying, etc. funktioniert.
[7] Nähere Erläuterungen dazu später, für den Algorithmus ist dies nicht direkt relevant.

brauchen.[8]
Unter der gleichen Annahme, zeigt sich, dass etwa 15! Operationen in einer Stunde ausgeführt werden können. Sprich es können $2 * (15 - 1) = 28$ Kisten in allen Möglichkeiten gepackt werden. (Unter der Annahme es gibt etwa x! Möglichkeiten für $2x$ Kisten[9])

Verkürzung der Laufzeit Eine Überlegung war, das Laufzeitverhalten durch Parallelisierung zu verkürzen. Allerdings verspricht dies aufgrund der hohen Laufzeitkomplexität von $O(n \cdot n!)$ kaum Abhilfe. Selbst bei 100 Kernen, sprich einer hundertfachen Beschleunigung[10] können gerade mal 17! Operationen ausgeführt werden. Das entspricht $2 * (17 - 1) = 32$ Kisten. Es können also 14%$(32/28 = 1.1428\dots)$ mehr Kisten gepackt werden, was nicht nennenswert viel ist.
Es kommt also für Frau Y. somit nicht in Frage die Packberechnung beispielsweise auf eine Rechnerfarm zu migrieren.

Problem Frau Y. hat also ihre mittlerweile 25 Kisten optimal packen können. Dadurch ist nun Platz in ihrem Keller frei geworden. Sie sieht es als ironisch an, dass sie genau deswegen nicht mehr Kisten in ihren Keller stellen kann, weil sie versucht den Platzverbrauch ihrer Kisten zu minimieren. Sprich, sie könnte beispielsweise eine Kiste direkt daneben stellen obwohl diese nicht mehr in die Berechnung einbezogen werden kann. Es ist offensichtlich, dass die ursprüngliche Motivation dadurch nicht erreicht wird. Wenn beispielsweise 200 Kisten gepackt werden sollen, bleiben 170 Kisten neben 30 optimal gepackten ungepackt.
Hierzu habe ich zwei Lösungsideen erstellt. Die grundlegende Motivation ist, die optimale Packung aufzugeben und stattdessen in menschlicher Zeit [11] trotzdem eine gute Packung auch zu einer großen Menge von Kisten zu finden.

1.3. Bruteforce nach Aufteilung

Ansatz Eine Möglichkeit wäre, den Bruteforce Algorithmus immer auf 30 Kisten anzuwenden und danach diese Kistensätze nebeneinander zu stellen.
Wichtig ist hierbei, dass die Kisten sinnvoll aufgeteilt werden, so dass jeder Kistenhaufen kleinere und größere Kisten hat um eine hohe Packdichte zu erreichen. ...

Laufzeitverhalten Dieser Algorithmus ist streng polynomiell. Genauer gesagt kann er sogar in $O(n \log n)$ Zeit ausgeführt werden. Dies ergibt sich aus einer Laufzeitanaylse des Algorithmus.
Zunächst müssen die Kisten sortiert werden in $O(n \log n)$. Das Aufteilen der Kisten in Gruppen braucht $O(n)$. Man erhält also $\frac{n}{30}$ Gruppen. Eine dieser zu packen geschieht in konstanter Zeit. (Auch wenn der Bruteforce ursprünglich eine Komplexität von $O(m \cdot m!)$ besitzt, ist mit $m = 30$ seine Laufzeit $O(30 \cdot 30!) = O(1)$, also konstant.) Daraus ergibt sich letzendlich

$$O(n \log n + n + \frac{n}{30} \cdot 1) = O(n \log n)$$

.

[8] Unter der Annahme dass das Prüfen und Hinzufügen einer Kiste in eine andere Kiste 1 ns dauert.
[9] Die Annahme zeigt sich als gar nicht so schlecht, wie man in **??** sieht
[10] Dies wird in der Praxis nie erreicht. Es muss immer ein gewisser Overhead für Synchronisation und Sequentielle Programmabläufe "geopfert" werden.
[11] menschliche Zeit = Lebenserwartung eines Menschen (75 Jahre)

1.4. Online Packer

Ansatz Ein etwas anderer Ansatz ist, Kistensätze inkrementiell zu erzeugen. Daher, es wird ein Algorithmus erfordert, welcher zu einem - mehr oder weniger gut - gepacktem Kistensatz und einer zu packenden Kiste ein neuen Kistensatz liefert, der, möglichst dicht gepackt, diese enthält. Eine weitere Beschränkung, die ich an den Algorithmus stelle, ist, dass er in höchstens $O(n)$ Zeit diesen neuen Kistensatz liefert. Hat man nun solch einen Algorithmus, lassen sich alle Kisten zusammen in $O(n \cdot n) = O(n)$ Zeit packen bei inkrementieller Kistensatzerzeugung.

Onlinealgorithmus Dieser Algorithmus kann von Frau Y. jedoch auch verwendet werden, um eine Kiste in der eine Lieferung verpackt war, in ihren vorhandenen Kistensatz hinzuzufügen. Es handelt sich also um einen Onlinealgorithmus. Die Entwicklung eines Onlinealgorithmus ist in der Aufgabenstellung weder explizit noch implizit gefordert. Es handelt sich also um eine eigenständige Erweiterung. Ich fande sie insofern sinnvoll, da sie ein Anwendungsfall direkt erfüllt, nämlich genau den, wenn Frau Y. eine einzelne Kiste erhält. Für Frau Y. ist es nun zwar möglich, einen neuen Kistensatz in linearer Zeit zu erhalten, aber es muss noch sichergestellt werden, dass auch ein möglicherweise nötiges Umpacken in linearer Zeit ausgeführt werden kann. Sprich, wir betrachten auch die "Laufzeit" Frau Y.'s und nicht die eines Computers. Für nachfolgende Strategien betrachte ich deswegen auch immer die Laufzeit für das Umpacken, welches nötig ist um die Kiste hinzuzufügen.

Strategien Es gibt unterschiedliche *Strategien* um einen Platz für eine Kiste in einem Kistensatz zu finden. Recht naheliegend sind unteranderem folgende.

FindeHalbleeren Findet eine KisteHalb die noch Platz für die neue Kiste bietet.

FindeGößerenLeeren Findet eine KisteLeer die noch Platz für die neue Kiste bietet.

FindeZwischenraum Findet eine Kiste die durch Umpacken einer Kind-Kiste in die neue Kiste genug Platz für die neue Kiste bietet.

FindeKleinereWurzel Findet eine Wurzel-Kiste die in die neue Kiste passt.

Offlinealgorithmus Werden die Kisten vor dem inkrementiellem Packen nach Volumen von groß nach klein sortiert, können Strategien 3 und 4 ohne Beschränkung weggelassen werden. Diese suchen nämlich Kisten, die kleiner sind als die hinzuzufügende, welche jedoch wegen der Sortierung nicht existieren können. Da jedoch zur Sortierung die Kisten bekannt sein müssen, handelt es sich nicht mehr um einen Online- sondern einen Offlinealgorithmus. Die Existenzberechtigung dieses Algorithmus ergibt sich aus der Tatsache, dass bessere Ergebnisse bei vorheriger Sortierung erhalten werden können als mit dem ursprünglichem Onlinealgorithmus. [12]

2. Implementierung

Zunächt wurde ein Kern implementiert, welcher Kisten und Kistensätze sinnvoll abbildet und hilfreiche Funktionen zur Operation auf diesen bietet. Die Datentypen wurden als unveränderbare Objekte implementiert um die Algorithmen zu vereinfachen.

[12] Siehe auch: 3.1

Kisten Es gibt drei Arten von Kisten: KisteLeer, KisteHalb und KisteVoll. Eine KisteLeer enthält keine weitere Kiste, eine KisteHalb enthält eine Kiste und eine KisteVoll enthält zwei. Es wurde zunächst ein **trait** Kiste implementiert, welches eine Anwendungsschnittstelle "nach außen" bietet und eine Schnittstelle "nach innen", welche von den drei Unterklassen implementiert werden muss. Das **trait** wurde als **sealed** implementiert, das heißt, nur Typen in der gleichen Datei dürfen dieses **trait** implementieren. Dies ermöglicht bessere Compilerunterstützung bei Pattern-matching, da bekannt ist, dass es nur genau 3 Unterklassen von Kiste gibt.
Neben der Implementierung von **val** hashCode: Int und **def** equals(Kiste): Kiste zur Verwendung als Hashkeys wurden auch oft verwendete anwendungspezifische Methoden implementiert. Zum einen existiert die Methode **def** +<(Kiste): Set[Kiste] welche alle Möglichkeiten **eine andere** Kiste in den durch **diese** Kiste definierten Kistenbaum gepackt werden kann zurückliefert. Weitere Methoden erwähne ich bei Benutzung.

Kistensatz Da eine Kiste die Wurzel eines Kistenbaums ist und diesen repräsentiert, (Betrachten Sie eine KisteLeer als einen ein-elementigen Kistenbaum), muss ein Kistensatz lediglich Referenzen auf die einzelnen Kiste-Objekte speichern. Es ergeben sich für die Datenstruktur, diesen Kistenwald (Menge von Kistenbäumen) zu verwalten, folgende drei Vorraussetzungen.

1. Das Ersetzen eines [Teil]baumes sollte möglichst billig sein. Da das Ersetzen einer Kiste als Löschen und anschließendes Hinzufügen dieser in den Baum implementiert ist, müssen sowohl die Lösch- als auch Hinzufügefunktionen schnelle Laufzeiten haben. Da die Datenstruktur unveränderbar ist, liefert jede Veränderung in dem durch eine Kiste repräsentiertem Kistenbaum eine neues Objekt zurück. Dieses Objekt muss dann durch eine Lösch- und eine Hinzufügeoperation in den Baum des KistenSatz aktualisiert werden.

2. "Duplikate" müssen zugelassen sein, da es passieren kann, das zwei Kisten genau die gleichen Maße haben.

3. Die Kistenbäume eines Kistensatzes müssen so sortiert sein, dass ein Vergleichen nach Elementen billig ist.

Diese drei Vorraussetzungen erfüllt meiner Ansicht nach ein geordneter Binärbaum am besten. Es wurde also die Scala Standardklasse TreeMap[Kiste,Int] verwendet. Sie bildet jeweils eine Kiste k auf eine Zahl $i_k > 1$ ab. Diese Zahlen sind gleich der Anzahl der einzelnen Kiste (bzw. Kistenbaum) in diesem Kistensatz. Somit erfolgt eine hinzufügen einer Kiste k (Scala: +(Kiste):Kistensatz mit dem Hinzufügen von $k \to 1$ in den Binärbaum, bzw. wenn k schon erhalten ist, mit dem Setzen von $k \to i_k + 1$. Analog erfolgt auch das Entfernen einer Kiste k (Scala: −(Kiste):Kistensatz), daher wenn k nicht enthalten oder $i_k = 1$ entferne k aus dem Binärbaum, andernfalls setze $k \to i_k - 1$.

Kistenpacker Da eine Menge von verschiedenen Algorithmen entwickelt wurden, bietete es sich an, von mehreren Algorithmen verwendete Funktionen auszulagern. Dies erfolgte in Scala durch Verwendung einer **trait**-Hierarchie.

3. Programmabläufe

3.1. Algorithm-Contest (Packdichte)

3.2. Algorithm-Contest (Laufzeit)

4. Programmtext

5. Programmnutzung

C. Zweite bearbeitete Aufgabe: (2) Containerklamüsel

1. Lösungsidee

1.1. Vorüberlegungen

Die Anordnung der Waggons zu den Container ist eine bijektive Abbildung von $[1, n]$ nach $[1, n]$, sprich, eine Permutation der Menge $[1, n]$. Jede Permutation lässt sich als Folge von disjunkten Zyklen darstellen.
"Eine Permutation π einer Menge wird *Zyklus* genannt, falls - grob gesprochen - die Elemente, die von π bewegt werden, zyklisch vertauscht werden. Genauer gesagt: Eine Permutation π heißt zyklisch, falls es ein $i \in X$ und eine natürliche Zahl k gibt, so dass die folgenden drei Bedingungen gelten:

1. $\pi^k(i) = i$,
2. die Elemente $i, \pi(i), \pi^2(i), \dots, \pi^{k-1}(i)$ sind paarweise verschieden,
3. jedes Element, das verschieden von $i, \pi(i), \pi^2(i), \dots, \pi^{k-1}(i)(= i)$ ist, wird von π fest gelassen.

Die kleinste natürliche Zahl k mit obiger Eigenschaft wird die *Länge* des Zyklus π genannt. Ein Zyklus der Länge k heißt auch k-Zyklus. Wir schreiben dann

$$\pi = (i\ \pi(i)\ \pi^2(i)\ \dots\ \pi^{k-1}(i)).$$

[..]

Darstellung einer Permutation als Produkt disjunker Zyklen. *Jede Permutation kann als Produkt zyklischer Permutationen geschrieben werden, von denen keine zwei ein Element gemeinsam haben.*
Das heißt: Zu jedem $\pi \in S_n$ gibt es zyklische Permutationen $\zeta_1, \dots, \zeta_s \in S_n$, so dass folgendes Eigenschaften erfüllt sind:
- $\pi = \zeta_1 \cdot \zeta_2 \cdot \dots \cdot \zeta_s$
- kein Element aus X, das als Komponente in ζ_i vorkommt, kommt in ζ_j vor ($i, j = 1, \dots, n, i \neq j$). (Das bedeuted: Wenn ein Element $x \in X$ in einem Zyklus ζ_i "vorkommt", so wird x von jedem anderen Zyklus ζ_j ($j \neq i$) fest gelassen.) "[13]

Die Darstellung der Permutation als Produkt disjunkter Zyklen erwies sich als günstig, denn nun kann das Problem in folgende zwei Teile aufgebrochen werden. Der erste ist, die Container eines Zyklus an die richtige Stelle zu bringen. Dies lässt sich relativ leicht realisieren, indem der Container am Anfang der Zyklen an die richtige Position gebracht wird, anschließend der zweite an die richtige, usw., bis der Ausgangspunkt wieder erreicht ist. Der zweite - etwas schwierige - Teil besteht darin, die Zyklenabarbeitung dort zu unterbrechen, wo eine andere beginnt.
Etwas anders ausgedrückt: Beginnt man an dem Anfang eines Zyklus, können dessen Container "in einem Stück" an die richtige Stelle gebracht werden und der Kran anschließend wieder an der Ausgangsposition angelangen. Wir werden etwas später sehen, dass dadurch tatsächlich auch immer ein optimaler Weg (zumindest innerhalb eines Zyklus) gefunden werden kann. Durch entsprechend richtige "Konkatenation" der einzelnen Befehlsketten für die einzelnen

[13] Definitionen, Sätze und Erklärung übernommen aus Lineare Algebra, Albrecht Beutelspacher

Zyklen lässt immer sich ein nach dem in der Aufgabenstellung vorgegebenem Gütekriterium optimaler Weg erstellen. Der durch Ausführung der durch einen Algorithmus berechneten Instruktionen abzufahrende Weg ist also minimal.

1.2. Datenstruktur

Permutationen können in einer indexierten Liste jeder Art (beispielsweise einem Array) gespeichert werden. Da in der Informatik jedoch indexierte Listen (insbesondere Arrays) meist Indizes aus $[0, n[$ besitzen muss dies beim Zugriff beachtet werden. Um also die Zahl p zu finden, auf die i durch $perm$ abgebildet wird, gilt $p = perm(i-1)$ jedoch nicht $p = perm(i)$. Es wird außerdem noch eine einfache Datenstruktur benötigt, um das Gleis mit Containerstellplätzen und Waggons abzubilden.
Diese ist in **??** genauer erläutert.

1.3. Ergebnisoptimaler Algorithmus

Entwurf Der Entwurf dieses Algorithmus' ergibt sich aus den obigen Überlegungen. Zunächst wird die Zerlegung in disjunkte Zyklen berechnet. Hierfür wird folgende Hilfsfunktion zur Berechnung *eines* Zyklus' verwendet. Wichtig ist hierbei zu beachten, dass die Waggonnummer an der Stelle idx durch perm(idx−1) dargestellt wird.

```scala
def cycle(perm: Seq[Int], start: Int): List[Int] = {
  def step(idx: Int): List[Int] =
    if(start == idx) Nil
    else idx :: step(perm(idx - 1))
  start :: step(perm(start -1))
}
```

Salopp gesagt, hangelt man sich so lange - bei einem Startindex beginnend - durch die Permutation, bis man wieder beim Anfangswert ankommt.
Nun lässt sich auch recht einfach ein Algorithmus zum Finden der disjunkten Zyklen einer Permutation p angeben. Die folgend dargestellte rekursive Funktion cyclesOf liefert eine Liste von disjunkten Zyklen (also eine Liste von Listen von Zahlen) die die Permutation darstellen. Um disjunkte Zyklen zu finden, müssen sich jeweils alle bisher abgearbeiteten Zahlen gemerkt werden. Dies erfolgt in einem Set (standardmäßig ein HashSet in Scala).
In jedem Rekursionsschritt wird zunächst der neue Startwert start gesucht. Der Startwert ist die erste Zahl von $1..n$ die noch nicht abgearbeitet wurde (also nicht in ready enthalten ist). Anschließend wird der neue Zyklus newCycle mit der Hilfsfunktion cycle berechnet. Dann wird die neue Menge aller abgearbeiteten Zahlen newReady gebildet, indem alle Zahlen aus newCycle in ready eingefügt werden. Zuletzt erfolgt der rekursive Aufruf, wobei newCycle vor den rekursiv berechneten Zyklen gespeichert wird. Die Rekursion wird abgebrochen, sobald alle Zahlen abgearbeitet wurden. Dies lässt sich daran erkennen, dass die Länge der Permutation gleich der Anzahl der abgearbeiteten Zahlen sind.

```scala
def cyclesOf(perm: Seq[Int], ready: Set[Int]): List[List[Int]] =
  if(ready.size == perm.length) Nil
  else {
    val start = (1 to perm.length) find (i => !ready.contains(i))
    val newCycle = cycle(perm, start)
    val newReady = ready ++ newCycle
    newCycle :: cyclesOf(perm, newReady)
  }
```

Optimale Ergebnisse Dieser Algorithmus liefert bereits optimale Ergebnisse im Sinne des Gütekriteriums der Aufgabenstellung Um dies zu zeigen, wird zunächst bewiesen, dass die Zyklen richtig gefunden werden.
Zunächst wird die Hilfsfunktion step der Funktion cycle auf Korrektheit geprüft. Da es sich um eine rekursive Funktion handelt, bietet sich hier Induktion an. Der Beweis, das die Hilfsfunktion step zu einer gegebenen Permutation $perm : [0, n[\to [1, n]$, einem Start- und Endwert start und einem Index idx eine Liste von Zahlen mit Länge n liefert, für die folgendes gilt. ...
Induktionsanfang:

...

Im folgenden können wir uns also der Korrektheit von cycle sicher sein. ...
Anschließend zeigen wir die Optimalheit vom eigentlichem Algorithmus, die Berechnung der Instruktionen.

Laufzeitverhalten Zunächst wird das Laufzeitverhalten des Algorithmus zum Finden der Zyklen analysiert. cyclesOf berechnet in jedem Schritt den neuen Startwert start. Dazu wird die Folge 1 bis zur Permutationslänge traversiert bis ein Wert gefunden wird der noch nicht abgearbeitet - sprich in ready enthalten - ist. Nimmt man an, dass das Prüfen auf Enthaltensein konstanten Zeitaufwand darstellt (Bsp. bei Verwendung eines HashSets), dann ergibt dies insgesamt eine Komplexität von $O(n)$. Die Berechnung eines Zyklus benötigt höchstens die Traversierung der Permutation, also ebenfalls $O(n)$. Anschließend werden die Zahlen, die im Zyklus enthalten sind, in ready eingefügt. Unter Annahme, dass wieder ein HashSet verwendet wird, ergibt das eine Komplexität von $O(n)$. Anschließend erfolgt der rekursive Aufruf. Sei c die Anzahl der Zyklen, dann wird cyclesOf c-mal aufgerufen. Die Laufzeitkomplexität zur Finden der Zyklen ist also $O(c \cdot n)$.

1.4. Optimaler Algorithmus

Das Laufzeitverhalten von $O(c \cdot n)$ ist zwar bereits recht gut, da die Anzahl der Zyklen im Normalfall nicht linear mit n steigen. (Eine zufällig erzeugte Permutation mit 10^7 Elementen hat meist weniger als 20 Zyklen)Der Worstcase bei $n/2$ Zyklen führt jedoch zu einer Worstcase-Komplexität von $O(n^2)$.
Deshalb soll als Erweiterung die Laufzeitkomplexität weiter verringert werden.
Außerdem sind die Algorithmen, wie sie oben angegeben sind, nicht tail-recursive. Das heißt bei jedem rekursivem Aufruf wird ein neuer Stack-frame allokiert. In der Praxis heißt dies, dass nur eine Rekursionstiefe von höchstens 10000 möglich ist.

Optimale Ergebnisse Wie oben (in 1.3) bereits gezeigt, können aus korrekten, sortierten Zyklen Instruktionen, die einen optimalen Weg für den Kran liefern, berechnet werden. Deshalb muss hier lediglich noch gezeigt werden, dass der neue Algorithmus wiederum korrekte und sortierte Zyklen berechnet.

Optimale Laufzeitkomplexität ...
Da jeder Container auf einen Waggon gebracht werden muss, muss für jeden Container mindestens ein Befehl erzeugt werden. Bei n Container sind dies also n Befehle. Das setzt einen Algorithmus mit einer Laufzeitkomplexität von mindestens $O(n)$ voraus. Der erstellte Algorithmus hat also **optimale Laufzeitkomplexität**.

2. Implementierung

Die Implementierung gliedert sich folgendermaßen.

Utils hilfreiche Methoden, unter anderem zur Zyklenfindung.

SlowUtils wie Utils, bloß werden die zuerst vorgestellten, langsameren Funktionen zur Zyklenfindung benutzt.

FastAlgorithm

SlowAlgorithm

2.1. "SlowAlgorithm"

2.2. "FastAlgorithm"

2.3. Randomisierte Permutationen

3. Programmabläufe

Bemerkung: Die Ausgaben der Konsole wurden per Hand nachformatiert zwecks besserem Einbetten in den Textfluss.

Beispiel aus der Aufgabenstellung Das Folgende zeigt den Ablauf der sich bei Eingabe des Beispiels aus der Aufgabestellung ergibt.
Zunächst wird die Permutation erzeugt und in perm gespeichert.

```
scala> val perm = Seq(4,3,2,1)
perm: Seq[Int] = List(4, 3, 2, 1)
```

Anschließend werden die Instruktionen erzeugt und in instrs gespeichert.

```
scala> val instrs = FastAlgorithm compute perm
instrs: Seq[de.voodle.tim.bwinf.container.Instruction] =
  Queue(TakeCon, MoveRight(1), Rotate, TakeCon, Rotate, PutCon,
        Rotate,  MoveRight(1), Rotate, PutWag,
        TakeCon, MoveLeft(1),  Rotate, PutWag,
        TakeCon, MoveRight(2), Rotate, PutWag,
        TakeCon, MoveLeft(3),  Rotate, PutWag, TakeCon)
```

Nun wird eine Maschine erzeugt, die die Instruktionen ausführen kann.

```
scala> val maschine = new Maschine(new Gleis(perm),true)
maschine: de.voodle.tim.bwinf.container.Maschine =
Container: 4 3 2 1
Waggons:   _ _ _ _
```

Zuletzt soll die Maschine die Instruktionen interpretieren.

```
scala> maschine interpret instrs
1 2 3 4
4 3 2 1
-->
  -->
  <--
  ----->
<-------
res0: de.voodle.tim.bwinf.container.Gleis =
Container: _ _ _ _
Waggons:   1 2 3 4
```

Bemerkenswert ist hier, dass der erstellte Algorithmus in diesem Fall exakt den gleichen Weg liefert wie im Beispiel der Aufgabenstellung angegeben. Es gibt noch verschiedene andere Wege. Beispielsweise kann das Prüfen auf überlappende Zyklen erst beim Zurückfahren geprüft werden, wodurch sich z.B. einer der folgender Ablauf ergeben könnte.

```
1 2 3 4              1 2 3 4
4 3 2 1              4 3 2 1
------->             ------->
     <--               <----
  <--                  -->
  -->                  <--
<----                <--
```

Zufällig erzeugte Permutation Ein nächstes - etwas größeres Beispiel ergibt sich aus zufälliger Erzeugung einer Permutation der Länge 20. Hierbei wird die Hilfsfunktion randPerm des Moduls Utils aufgerufen und das Ergebnis wie vorher in perm gespeichert.

```
scala> val perm = Utils randPerm 20
perm: Array[Int] = Array(20, 11, 2, 8,  1, 16, 10, 17, 19, 14,
                          5, 12, 9, 3, 13, 15, 18,  4,  7,  6)
```

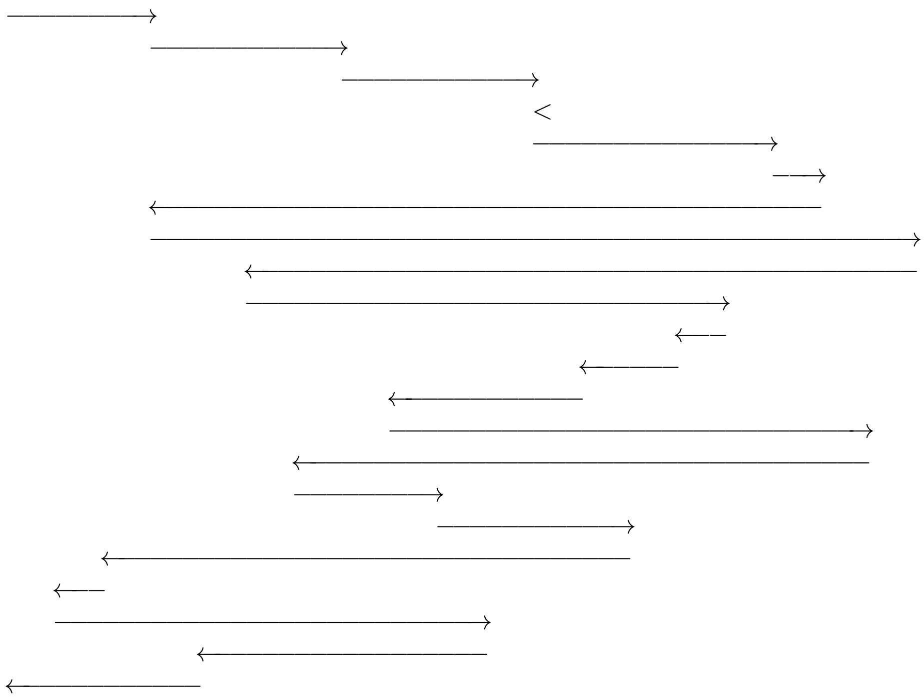
Anschließend werden wieder die Instruktionen mit der Funktion compute des Moduls FastAlgorithm berechnet und in instrs gespeichert.

```
scala> val instrs = FastAlgorithm compute perm
instrs: Seq[de.voodle.tim.bwinf.container.Instruction] =
  Queue(TakeCon, MoveRight(3),  Rotate, TakeCon, Rotate, PutCon,
        Rotate,  MoveRight(4),  Rotate, PutWag,
        TakeCon, MoveRight(4),  Rotate, TakeCon, Rotate, PutCon,
        Rotate,  MoveLeft(0),   Rotate, PutWag,
        TakeCon, MoveRight(5),  Rotate, PutWag,
        TakeCon, MoveRight(1),  Rotate, PutWag, T...
```

Zuletzt wird wieder eine Maschine maschine erzeugt um die Instruktionen zu interpretieren.

```
scala> val maschine = new Maschine(new Gleis(perm),true)
maschine: de.voodle.tim.bwinf.container.Maschine =
Container: 20 11 2 8 1 16 10 17 19 14 5 12 9 3 13 15 18 4 7 6
Waggons:    _  _ _ _ _  _  _  _  _  _ _  _ _ _  _  _  _ _ _ _

scala> maschine interpret instrs
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
20 11  2  8  1 16 10 17 19 14  5 12  9  3 13 15 18  4  7  6
```



```
res0: de.voodle.tim.bwinf.container.Gleis =
Container: _ _ _ _ _ _ _ _ _ _  _  _  _  _  _  _  _  _  _  _
Waggons:   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

TODO: Interpretation!

Demonstration der Skalierbarkeit Nun soll die Skalierbarkeit demonstriert werden, die als Erweiterung in Form von Tail-rekursiven Funktionen und linearer Laufzeitkomplexität implementiert wurde.
Hierfür erzeugen wir eine zufällige Permutation von 6,4 Millionen ($6,4*10^6$) Zahlen, die unsere Container darstellt. Anschließend werden wie oben auch, die Instruktionen berechnet und interpretiert.
Für Demonstrationszwecke wird außerdem die benötigte Zeit für jeden Schritt berechnet. Dies hat nicht das Ziel genaue Benchmarkwerte zu liefern, sondern vielmehr einen Anhaltspunkt für das Laufzeitverhalten. Hierfür wurde ein kleines Scala Programm geschrieben, zu finden im Modul Utils.
Nun wird zunächst das Programm ausgeführt. (Die Permutation wird nicht ausgegeben, um Platz zu sparen.)

```
scala> val verified = Utils demonstrate 6400000
Time used for computing Cycles: 30093
Number of cycles: 18
Time used: 110879
Time used interpreting: 10639
verified: Boolean = true
```

Interessant ist hier die Beobachtung, dass es nur 18 Zyklen gibt, bei einer Permutationslänge von 10^7. TODO: !!! Interpret more !!!

4. Programmnutzung

5. Programmtext