

29. Bundeswettbewerb Informatik 2010/2011

2. Runde

Tim Taubner, Verwaltungsnummer 29.0108.01

14. 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 jede Aufgabe werden jeweils die Lösungsidee und entwickelte Algorithmen und Datenstrukturen, sowie geeignete Programm-Ablaufprotokolle, wie die Programmnutzung funktioniert und der Programm-Text selbst. Die interaktiv ausführbaren Lösungen werden in der Dokumentation erklärt. Der Quelltext ist beigefügt. Die Implementierung ist ebenfalls dokumentiert und gibt grobe Überblicke über die Strukturierung des Programms und der einzelnen Module/Klassen. In diesem ist sowohl kurz die Idee als auch die Implementationerläuterung enthalten.

Erweiterungen und weiterführende Gedankengänge, die über die direkte Aufgabenstellung hinausgehen, sind ebenso dokumentiert. Diese werden jeweils entsprechend gekennzeichnet.

Sämtliche Dokumentation und alle ausführbaren Erzeugnisse sind auf einer separaten CD mitgeliefert. Wie die erstellten Programme von der mitgelieferten CD aus gestartet werden können, ist im ersten Abschnitt, Allgemeines, erläutert. Alle eingereichten Quelldateien, Kunsterzeugnisse (wie z.B. Bilder) und ausführbare Programmdistributionen wurden alleine von mir - Tim Taubner - erstellt.

Inhaltsverzeichnis

A Allgemeines	2
B Erste bearbeitete Aufgabe: (1) Kisten in Kisten in Kisten	4
C Zweite bearbeitete Aufgabe: (2) Containerklamüsel	9

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 passable Dokumentation erstellt werden sollte. 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. Aufgabe¹ “informatische Kunst”. Viel anschaulicher lässt sich nicht zeigen, dass Informatik *mehr* als nur Programmieren ist.

Wahl der Programmiersprache Die von mir zur Aufgabenlösung benutzte Programmiersprache ist durchgehend Scala. Ich bevorzuge Scala, weil es mit ihr möglich ist, knappen und kompakten 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. Trotzdem hoffe ich, dass der moderne und aussagekräftige Code allgemein für Informatiker gut lesbar ist. Deswegen habe ich auch auf Pseudo-Code verzichtet. Diese zusätzliche Sprache und Syntax neben Scala und Mathematik stellt eine zusätzliche Komplexität. Stattdessen formuliere ich alle Algorithmen in Scala-Code und erläutere diesen.

Danksagungen Ich erlaube mir hier, Personen zu danken, die mir zu der vorliegenden Arbeit verholfen haben. Auch wenn alle Leistungen im Sinne des Wettbewerbs von mir erbracht wurden, habe ich dazu viel Unterstützung meiner Freundin und meiner Familie erhalten.

2 Dateistruktur der CD

Die Dateien auf der CD sind wie folgt strukturiert. Zu jeder Aufgabe wurde ein Ordner **AufgabeX** mit den zwei folgenden Unterordnern angelegt.

src Unterordner, in dem die Quelltexte liegen

dist Unterordner, in dem ausführbare Dateien 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 L^AT_EX-Quelldateien, mit der diese Dokumentation erzeugt wurde sind 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

¹siehe Einsendung zur 1. Runde, Einsendungsnr. 108

Grafikkarte: beliebig

Getestete Betriebssysteme: 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)

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 Betriebssystemen starten können.²

Unter Windows Unter Windows starten Sie die Kommandozeile durch: Start → Ausführen → 'cmd' eingeben → Kommandozeile. Nun können Sie durch Angabe des Laufwerksbuchstabens 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 Standard-distributionen 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 Standard-distributionen 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)

²Ich 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

Inhalt der Aufgabendokumentation

1	Lösungsidee	4
2	Algorithmen und Datenstrukturen	4
2.1	Datenstrukturen	4
2.2	Brute-force	4
2.3	Brute-force nach Aufteilung	6
2.4	Online Packer	6
3	Implementierung	7
4	Programmabläufe	8
4.1	Algorithm-Contest (Packdichte)	8
4.2	Algorithm-Contest (Laufzeit)	8
5	Programmtext	8
6	Programmnutzung	8

1 Lösungsidee

2 Algorithmen und Datenstrukturen

2.1 Datenstrukturen

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, \dots, k_n ein Kistensatz ks mit den Wurzeln w_1, \dots, w_m zu erzeugen mit $elems(ks) = \{k_1, k_2, \dots, k_n\}$. Seien v_1, \dots, v_m die jeweiligen Volumina der Wurzeln w_1, \dots, w_m . Dann gilt es als weitere Aufgabe $\sum_{i=1}^m v_i$ zu minimieren.

2.2 Brute-force

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.³

Wichtig ist, um den Code zu verstehen, dass $(satz ++< kiste)$ alle Möglichkeiten erzeugt, wie man die Kiste in einen Satz einfügen kann.⁴

³Beispielsweise sollten Sie wissen, wie `foldLeft`, `currying`, etc. funktioniert.

⁴Nähere Erläuterungen dazu später, für den Algorithmus ist dies nicht direkt relevant.

```

1 // Nacheinander die Kisten "auffalten" mit Hilfe der hilfsPacken Funktion
2 def packe = (Set[KistenSatz]() /: kisten) ( hilfsPacken )
3 def hilfsPacken(sätze: Set[KistenSatz], kiste: Kiste) =
4   if(sätze.isEmpty)
5     Set(KistenSatz(kiste :: Nil)) // KistenSatz nur mit der Kiste kiste
6   else
7     (Set[KistenSatz]() /: sätze) { // Beginne mit leerer Menge
8       (menge, satz) =>
9         menge ++ // Füge neue Möglichkeiten der menge hinzu
10        (satz ++< kiste) // Erzeugt neue Möglichkeiten
11      }

```

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 $n! + n \cdot (n-1)! + (n-1) \cdot (n-2)! + \dots + 2 \cdot 1! + 1$. 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 brauchen.⁵ Unter der gleichen Annahme, zeigt sich, dass etwa 15! Operationen in einer Stunde ausgeführt werden können. Sprich es können $2 \cdot (15-1) = 28$ Kisten in allen Möglichkeiten gepackt werden. (Unter der Annahme es gibt etwa $x!$ Möglichkeiten für $2x$ Kisten⁶)

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 Beschleunigung⁷ können gerade mal 17! Operationen ausgeführt werden. Das entspricht $2 \cdot (17-1) = 32$ 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 Zeit⁸ trotzdem eine gute Packung auch zu einer großen Menge von Kisten zu finden.

⁵Unter der Annahme dass das Prüfen und Hinzufügen einer Kiste in eine andere Kiste 1 ns dauert.

⁶Die Annahme zeigt sich als gar nicht so schlecht, wie man in ?? sieht

⁷Dies wird in der Praxis nie erreicht, es muss immer ein gewisser Overhead für Synchronisation und sequentielle Programmabläufe "geopfert" werden.

⁸menschliche Zeit << Lebenserwartung eines Menschen (75 Jahre)

2.3 Brute-force nach Aufteilung

Ansatz Eine Möglichkeit wäre, den Bruteforce Algorithmus immer auf einen Teil der Kisten anzuwenden und danach diese so erzeugten Kistensätze nebeneinander zu stellen. Wenn tf die Anzahl Kisten pro Gruppe sein soll, so wird jede tf -ste Kiste in eine Gruppe zugeordnet.

Laufzeitverhalten Dieser Algorithmus ist streng polynomiell. Genauer gesagt kann er sogar in linearer Zeit ausgeführt werden. Dies ergibt sich aus einer Laufzeitanalyse des Algorithmus. Sei im folgenden tf der Teilungsfaktor, also die Anzahl Kisten die jeweils eine Gruppe bilden. Das Aufteilen der Kisten in Gruppen braucht $O(n)$. Man erhält also $\frac{n}{tf}$ Gruppen. Eine dieser zu packen geschieht in konstanter Zeit. (Auch wenn der Bruteforce ursprünglich eine Komplexität von $O(m \cdot m!)$ besitzt, ist mit $m = tf$ seine Laufzeit $O(tf \cdot tf!) = O(1)$, also konstant, bei konstantem tf .) Daraus ergibt sich letztendlich

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

. Diese Laufzeitkomplexität ist sogar optimal, denn es müssen in jedem Fall für jede Kiste Laufzeit “verbraucht” werden bei der Ausgabe der Packart.

2.4 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 $O(n \cdot n) = O(n^2)$ 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 fand 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 unter anderem 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 inkrementiellen 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.⁹

3 Implementierung

Zunächst 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.

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 Compiler-unterstützung 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 anwendungsspezifische 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, die den Kistenwald (Menge von Kistenbäumen) verwaltet folgende drei Voraussetzungen.

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 kurze Laufzeiten haben. Da die Datenstruktur unveränderbar ist, liefert jede Veränderung in dem durch eine Kiste repräsentiertem Kistenbaum ein 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, dass 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 Voraussetzungen erfüllt meiner Ansicht nach ein geordneter Binärbaum am besten. Es wurde also die Scala Standardklasse TreeMap[Kiste,Int] verwendet. Sie bildet jeweils eine

⁹Siehe auch: 4.1

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 $k \rightarrow 1$ in den Binärbaum, bzw. wenn k schon erhalten ist, mit dem Setzen von $k \rightarrow i_k + 1$. 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 $k \rightarrow i_k - 1$.

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

4 Programmabläufe

4.1 Algorithm-Contest (Packdichte)

4.2 Algorithm-Contest (Laufzeit)

5 Programmtext

6 Programmnutzung

C Zweite bearbeitete Aufgabe: (2) Containerklamüsel

Inhalt der Aufgabendokumentation

1	Lösungsidee	9
2	Algorithmen und Datenstrukturen	11
2.1	Datenstruktur	11
2.2	Ergebnisoptimaler Algorithmus	11
2.3	Ergebnis- und Laufzeitoptimaler Algorithmus	19
3	Implementierung	21
3.1	Cycler - Berechnung der Zyklen	21
3.2	Instructor - Berechnung der Instruktionen	21
3.3	Gleis - Speichern des Zustand	22
3.4	Maschine - Interpretieren der Instruktionen	22
3.5	ListBuffer - Erweiterung einer Standardklasse	22
3.6	Utils - Helfende Methoden	22
4	Programmabläufe	24
4.1	Beispiel aus der Aufgabenstellung	24
4.2	Zufällig erzeugte Permutation	26
4.3	Demonstration der Skalierbarkeit	27
5	Programmnutzung	27
5.1	Permutationen erzeugen	27
5.2	Erzeugen der Instruktionen	28
5.3	Simulation der Maschine	28
5.4	Zeitmessung	29
6	Programmtext	30
6.1	Cycler	30
6.2	Instructor	31
6.3	Gleis	32
6.4	Maschine	33
6.5	Instructions	34
6.6	Utils	34
6.7	ListBuffer	35

1 Lösungsidee

Für die Lösung der Aufgabe werden bestimmte Eigenschaften von Permutationen zu Nutze gemacht. Zunächst lässt sich feststellen, dass die Anordnung der Waggon zu den Containern eine bijektive Abbildung von $[1, n]$ nach $[1, n]$, sprich, eine Permutation der Menge $[1, n]$ ist. Die entscheidende Eigenschaft die der von mir entwickelte Algorithmus nutzt ist die Tatsache, dass sich jede Permutation als Folge von disjunkten Zyklen darstellen lässt.

Was ein Zyklus im Ungefähren ist und was er für die Aufgabe bedeutet, lässt folgende Darstellung vermuten. Man beachte, dass die Container “in einem Stück” getauscht und an die richtige Position gebracht werden können.

```

1 1 2 3 4 5 6 7 8 (Index)
2 2 5 3 8 4 6 7 1 (Containernummer)
3 -->
4 ----->
5      <--
6      ----->
7 <-----

```

Um den Begriff eines Zyklus' genauer einzuführen, zitiere ich folgend Beutelspacher.

“Eine Permutation π von X wird ein *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 $i, \pi(i), \pi^2(i), \dots, \pi^{k-1}(i), \pi^k(i)(= i)$ 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 disjunkter Zyklen. *Jede Permutation kann als Produkt zyklischer Permutationen geschrieben werden, von denen keine zwei ein Element gemeinsam haben.*

Das heißt: Zu jedem $\pi \in S_n$ [S_n ist die Menge aller Permutation von $[1, n]$ in sich.] gibt es zyklische Permutationen $\zeta_1, \dots, \zeta_s \in S_n$, so dass folgende Eigenschaften erfüllt sind:

- $\pi = \zeta_1 \cdot \zeta_2 \cdot \dots \cdot \zeta_s$
- kein Element aus X [$X = [1, n]$], das als Komponente in ζ_i vorkommt, kommt in ζ_j vor ($i, j = 1, \dots, s, i \neq j$). (Das bedeutet: 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.)”¹⁰

Die Darstellung der Permutation als Produkt disjunkter Zyklen erweist sich als günstig, denn nun kann das Problem in folgende zwei Teile aufgebrochen werden. Der erste ist, die Container eines Zyklus' an die richtigen Stellen zu bringen. Dies lässt sich relativ leicht realisieren, indem der Container am Anfang eines Zyklus' an die richtige Position gebracht wird, anschließend der zweite an die richtige Position, ..., bis der Ausgangspunkt wieder erreicht ist. Der zweite Teil besteht also darin, die Zyklenabarbeitung dort zu unterbrechen, wo eine andere beginnt. Da nach der Abarbeitung des nächsten Zyklus der Kran wieder an der Position ist, wo der erste Zyklus unterbrochen wurde, kann die Abarbeitung “einfach” fortgesetzt werden.

Etwas anders ausgedrückt: Beginnend am Anfang eines Zyklus', können dessen Container “in einem Stück” an die richtige Stelle gebracht werden und der Kran kann anschließend wieder an der Ausgangsposition ankommen. Wir werden etwas später sehen, dass dadurch tatsächlich auch immer ein optimaler Weg (zumindest innerhalb eines Zyklus) gefunden werden kann.

¹⁰Definitionen, Sätze und Erklärung übernommen aus Lineare Algebra, Albrecht Beutelspacher, S.174f

Durch entsprechend richtige “Konkatenation” bzw. “Verschachtelung” der Befehlsketten für die einzelnen Zyklen lässt sich immer ein nach dem in der Aufgabenstellung vorgegebenem Gütekriterium optimaler Weg des Krans erstellen. Der durch Ausführung der berechneten Instruktionen abzufahrende Weg ist also minimal.

Vor der folgenden, eigentlichen Dokumentation möchte ich noch von mir verwendete Konventionen erläutern. Betrachte ich in einem Teil der Dokumentation Code, bzw. Codeausschnitte, benutze ich `monospaced Font` und die im Code selber benutzten Bezeichner um diese darzustellen.

Zur Erläuterung mathematischer Überlegungen wird der *math-Mode* von \TeX benutzt. Statt beispielsweise `perm` für eine Permutation oder `cycle` für einen oben beschriebenen Zyklus als Bezeichner im “code-Mode” benutze ich π respektive ζ im “math-Mode”. Es werden also die jeweils passenderen Bezeichner und Symbole verwendet.

2 Algorithmen und Datenstrukturen

2.1 Datenstruktur

Permutationen auf $[1, n]$ 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. Der Definitionsbereich ist also um eins “nach links” verschoben. Um also die Zahl p zu finden, auf die i durch `perm` abgebildet wird, gilt $p = \text{perm}(i-1)$ und nicht $p = \text{perm}(i)$.

Es wird außerdem noch eine einfache Datenstruktur benötigt, um das Gleis mit Containerstellplätzen und Waggonen abzubilden. Hierfür werden zwei Arrays verwaltet, die zu jedem Index den Container speichern der gerade auf dem Containerstellplatz bzw. Waggon steht. Auch hier sind die Indizes um “eins nach links” verschoben. Die Implementierung dieser Datenstruktur wird in 3.3 noch genauer erläutert.

2.2 Ergebnisoptimaler Algorithmus

In diesem Abschnitt wird zunächst ein Algorithmus entworfen, der optimale Ergebnisse (im Sinne von kürzesten Kranwegen) berechnet. Anschließend wird das Laufzeitverhalten dieses Algorithmus betrachtet, welches gut, jedoch nicht bestmöglich ist. Im darauffolgendem Abschnitt wird ein Algorithmus vorgestellt, der sowohl optimale Ergebnisse berechnet als auch optimale Laufzeitkomplexität vorweist.

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.

Salopp gesagt, handelt man sich so lange - bei einem Startindex beginnend - durch die Permutation, bis man wieder beim Anfangswert ankommt. Genauer betrachtet, liefert die Unterfunktion `step` die verbleibenden Zahlen des Zyklus’ nach `idx`. `step` bricht mit der leeren Liste ab, wenn `start` wieder erreicht wird. Andernfalls reicht `step` den aktuellen Wert `idx` vor die restlichen - rekursiv durch `step` - berechneten Zahlen. `cycle` braucht nur mehr `step` mit `start` aufzurufen. Das Ergebnis von `cycle` ist also ein Zyklus der oben beschriebenen Form $i, \pi(i), \pi^2(i), \dots, \pi^{k-1}(i)$ als `start :: perm(start-1) :: perm(perm(start-1)-1) :: ... :: Nil`, wichtig ist hierbei zu beachten, dass die Waggonnummer an der Stelle `idx` durch `perm(idx-1)` dargestellt

wird.

```

1 def cycle(perm: Seq[Int], start: Int): List[Int] = {
2   def step(idx: Int): List[Int] =
3     if(start == idx) Nil
4     else idx :: step(perm(idx - 1))
5   step(start)
6 }

```

Nun lässt sich auch recht einfach ein Algorithmus zum Finden der disjunkten Zyklen einer Permutation 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 jeweils alle bisher abgearbeiteten Zahlen gespeichert werden. Dies erfolgt in einem `Set` (standardmäßig ein `HashSet` in Scala).

In jedem Rekursionsschritt wird zunächst der neue Startwert gesucht. Dieser ist die erste Zahl von 1 bis `perm.length` die noch nicht abgearbeitet wurde, also die nicht in `handled` enthalten ist. Wurde ein Startwert `start` gefunden, dann wird anschließend der neue Zyklus `newCycle` mit der Hilfsfunktion `cycle` berechnet. Zudem wird die neue Menge aller abgearbeiteten Zahlen `newHandled` gebildet, indem alle Zahlen aus `newCycle` in `handled` eingefügt werden. Zuletzt erfolgt der rekursive Aufruf, wobei `newCycle` vor den rekursiv berechneten Zyklen angefügt wird. Wurde jedoch kein Startwert gefunden, so wird die Rekursion abgebrochen. Es wird dann die leere Liste `Nil` zurückgegeben.

```

1 def cyclesOf(perm: Seq[Int], handled: Set[Int]): Cycles =
2   (1 to perm.length) find (i => !handled.contains(i)) match {
3     case Some(start) =>
4       val newCycle = cycle(perm, start)
5       val newHandled = handled ++ newCycle
6       newCycle :: cyclesOf(perm, newHandled)
7     case None =>
8       Nil
9   }

```

Anhand der berechneten Zyklen wird im nächsten Schritt die Instruktionskette errechnet. Hierfür wird zunächst die folgend abgebildete Methode `computeFromCycles` definiert, welche sich einer - gleich anschließend betrachteten - weiteren Funktion `computeCycle` bedient.

```

1 def computeFromCycles(cycles: Cycles): Seq[Instruction] = {
2   // Füge TakeCon hinzu, damit bereits ein Container auf dem Kran ist;
3   // Lösche letztes Element (PutWag) mit init
4   TakeCon :: computeCycle(cycles.head, cycles.tail).init._1.toList
5 }

```

Im Allgemeinen soll die Funktion `computeCycle` für einen Zyklus `cycle` und die restlichen Zyklen `other` die Instruktionen für einen Weg liefern, so dass alle Elemente der gegebenen Zyklen an die richtige Position gebracht werden und der Kran wieder an die Startposition gebracht wird. Hierbei geht `computeCycle` davon aus, dass bereits der 1. Container des Zyklus auf den Kran gehoben wurde und noch kein anderer Container des Zyklus' bewegt wurde. Außerdem werden Container immer auf der Containerseite transportiert. Das Grundprinzip des Algorithmus' ist es, dass nacheinander alle Elemente des Zyklus abgearbeitet werden. Dazu wird ein `foldLeft` über den Zyklus ausgeführt. Damit der Kran auch wieder zum ersten Element

(Startposition) zurückgefahren wird, wird dieser Startwert `first` an den Zyklus angehängt. In jedem Schritt werden die bisherigen Instruktionen `instrs`, die verbleibenden Nachfolgerzyklen `cyclesLeft` und das vorherige Element `prev` an die Hilfsmethode `step` übergeben.

Im folgenden ist der Code dargestellt nur mit Deklaration aber ohne Definition der Hilfsfunktion `step`. (Ich entschied mich zwecks Lesbarkeit und Strukturierung, den Code in mehrere Teile aufzuteilen.) Bemerkung: Um den Code kompakt zu halten, wurden die sogenannten “type aliases” `Cycle` für `List[Int]`, `Cycles` für `List[Cycle]` und, speziell für die Hilfsfunktion `step` der Alias `Step` für das Tuple `(ListBuffer[Instruction], Cycles, Int)` benutzt.

```

1 def computeCycle(cycle: Cycle,
2                 other: Cycles): (ListBuffer[Instruction], Cycles) = {
3   val max = cycle.max
4
5   type Step = (ListBuffer[Instruction], Cycles, Int)
6   def step(instrs: ListBuffer[Instruction], cyclesLeft: Cycles,
7           prev: Int, cur: Int): Step = [...]
8
9   val erster = cycle.head
10  val initial = (ListBuffer[Instruction](), other, erster)
11  // Arbeite alle Elemente des Zyklus' ab
12  val (instrs, cyclesLeft, last) =
13    (cycle.tail :+ erster).foldLeft(initial) {
14      case ((instrs, cyclesLeft, prev), cur) =>
15        step(instrs, cyclesLeft, prev, cur)
16    }
17  (instrs, cyclesLeft)
18 }

```

Die Hilfsfunktion `step` unterscheidet 3 Fälle (diese sind hinter den `case` Anweisungen in Klammern in den Kommentaren im nachfolgendem Codeausschnitt markiert).

1. Wenn das letzte betrachtete Element `prev` das Maximum `max` ist und es ein nächsten Zyklus `nextCycle` gibt, dessen erstes Element `next` eins weiter rechts von `max` bzw. `prev` ist, dann konkateniere die Zyklen entsprechend. Das heißt, es wird erst mit `computeCycle` die Instruktionen `cycleInstrs` für die nächsten Zyklen berechnet und diese anschließend angehängt. Zudem müssen ein paar wenige Instruktionen “zwischen” den Zyklen, also vor und nach `cycleInstrs` generiert werden. Anschließend wird `step` nochmals aufgerufen, diesmal mit den neuen Instruktionen `extraInstrs` und keinen restlichen Zyklen, da diese bereits alle abgearbeitet sind.
2. Wenn das aktuell betrachtete Element `cur` größer als das erste Element `next` des nächsten Zyklus `nextCycle` ist, dann wird zunächst der nächste Zyklus abgearbeitet. Hierfür wird `computeCycle` mit `nextCycle` und den restlichen Zyklen `cyclesLeft.tail` aufgerufen. Hierbei können Zyklen “übrig” bleiben, nämlich wenn das maximale Element des Zyklus’ `next` kleiner ist als das maximale Element `max` dieses Zyklus’ `cycle`. Diese möglicherweise “übrig” geliebenen Zyklen `newCyclesLeft` werden zusammen mit den neuen Instruktionen wieder an `step` übergeben.
3. Wenn weder der 1. noch der 2. Fall zutrifft, werden lediglich Instruktionen generiert, die den Kran von `prev` nach `cur` bewegen, aktuellen Container auf den Waggon ablegen und dann den Container auf dem Containerstellplatz aufheben.

Anschließend werden die durch das `foldLeft` erzeugten Instruktionen und restlichen Zyklen zurückgegeben.

Folgend ist der Scalacode abgebildet, welcher die Hilfsfunktion `step` darstellt. Dieser Codeausschnitt ist deutlich komplexer als die vorherigen. Deswegen wurden Kommentare und Markierungen hinzugefügt.

```

1 type Step = (ListBuffer[Instruction], Cycles, Int)
2
3 def step(instrs: ListBuffer[Instruction], cyclesLeft: Cycles,
4         prev: Int, cur: Int): Step =
5   cyclesLeft.headOption match {
6     // Gibt es ein nächsten Zyklus direkt nach diesem beginnend?
7     // ===== (1) =====
8     case Some(nextCycle @ (next :: _)) if prev == max && max + 1 == next =>
9       // Wenn ja, "konkateneriere" diese.
10      val (cycleInstrs, _) =
11        computeCycle(cyclesLeft.head, cyclesLeft.tail)
12      val extraInstrs = instrs ++=
13        ListBuffer(PutCon, MoveRight, TakeCon) ++=
14        cycleInstrs ++= ListBuffer(MoveLeft, TakeCon)
15      step(extraInstrs, Nil, prev, cur)
16    // Gibt es ein nächsten Zyklus
17    // und beginnt dieser vor dem nächsten Element?
18    // ===== (2) =====
19    case Some(nextCycle @ (next :: _)) if cur > next =>
20      // Wenn ja, dann arbeite erst nextCycle ab.
21      val (cycleInstrs, newCyclesLeft) =
22        computeCycle(nextCycle, cyclesLeft.tail)
23      val newInstrs = instrs ++=
24        ListBuffer(Move(prev -> next),
25                  Rotate, TakeCon,
26                  Rotate, PutCon, Rotate) ++= cycleInstrs
27      step(newInstrs, newCyclesLeft, next, cur)
28    // ===== (3) =====
29    case _ =>
30      // Andernfalls, fahre einfach mit der Abarbeitung fort.
31      val newInstrs = instrs ++=
32        ListBuffer(Move(prev -> cur), Rotate, PutWag, TakeCon)
33      (newInstrs, cyclesLeft, cur)
34  }

```

Optimale Ergebnisse Dieser Algorithmus liefert bereits optimale Ergebnisse im Sinne des Gütekriteriums der Aufgabenstellung. Um dies zu zeigen, wird bewiesen, dass die Zyklen richtig gefunden werden.

Zunächst wird die Korrektheit der Hilfsfunktion `cycle` gezeigt. Das heißt, wir vergewissern uns, dass `cycle` zu einer gegebenen Permutation `perm` immer den Zyklus findet, der an dem Startindex `start` beginnt. Da ich nachfolgend nun mathematisch Argumentieren möchte, ersetze ich die Programmbezeichnungen durch mathematische. Konkret stelle ich `perm` durch π , den gesuchten Zyklus durch ζ und `start` durch i dar. Es ist also ein Zyklus ζ der folgenden Form gesucht.

$$\zeta = (i, \zeta(i), \zeta^2(i), \dots, \zeta^{k-i}(i))$$

Für alle x die im Zyklus ζ enthalten sind gilt $\zeta(x) = \pi(x)$. Weiter sind genau die Elemente

$i, \zeta(i), \zeta^2(i), \dots, \zeta^{k-i}(i)$ in ζ enthalten, also gilt

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

Nun betrachten wir nochmals die Funktionsweise von `cycle`, bzw. von `step`. Wir behaupten zunächst `step` liefert zu einer Zahl $j = \pi^x(i)$ die Zahlen $\pi^x(i), \pi^{x+1}(i), \dots, \pi^{k-1}(i)$. Dies machen wir uns durch Induktion über x klar. Sei also $x = k$. Dann gilt nach Definition eines Zyklus' $j = \pi^x(i) = \pi^k(i) = i$, also bricht `step` hier ab und liefert die leere Liste, was in der Tat korrekt ist. Nun können wir annehmen, `step` liefert für ein $j = \pi^x(i)$ bereits die Zahlen $\pi^x(i), \pi^{x+1}(i), \dots, \pi^{k-1}(i)$. Also zeigen wir nun, dass `step` auch für ein $l = \pi^{x-1}(i)$ die richtigen Zahlen liefert. `step` reiht also l vor die Zahlen, die durch Aufruf von `step` mit $\pi(l) = \pi(\pi^{x-1}(i)) = \pi^x(i) = j$ berechnet werden. Das ergibt genau die Zahlen $\pi^{x-1}(i), \pi^x(i), \pi^{x+1}(i), \dots, \pi^{k-1}(i)$. Die Aussage ist somit bewiesen. Wird nun `step` - wie in `cycle` - mit $j = \pi^0(i) = i$, also $x = 0$ aufgerufen, erhalten wir korrekterweise die Zahlen

$$(\pi^x(i), \pi^{x+1}(i), \dots, \pi^{k-1}(i)) = (\pi^0(i), \pi^1(i), \dots, \pi^{k-1}(i)) = (i, \pi(i), \pi^2(i), \dots, \pi^{k-1}(i)) = \zeta.$$

Im Folgenden können wir uns also der Korrektheit von `cycle` sicher sein. Nun soll die Korrektheit von `cyclesOf` gezeigt werden. Auch hier wähle ich mathematische Symbole/Bezeichner. Die Liste der disjunkten Zyklen, die `cyclesOf` berechnen soll bezeichne ich mit $\zeta_1, \zeta_2, \dots, \zeta_o$, die Menge aller fertigen Elemente `handled` mit Z . Wir wollen also beweisen, dass `cyclesOf` zu einer gegebenen Permutation π und einer leeren Menge von "fertigen" Elementen eine Liste von disjunkten Zyklen $\zeta_1, \zeta_2, \dots, \zeta_o$ zurückgibt, wobei o die Anzahl disjunkter Zyklen ist und $x < y \Leftrightarrow \min(\zeta_x) < \min(\zeta_y)$ für alle $x, y = 0 \dots o$. Es sollen also nach Startwert sortierte Zyklen zurückgeliefert werden. Es wird im folgenden wieder Induktion verwendet. Im Induktionsanfang soll also gezeigt werden, dass `cyclesOf` für $Z = \zeta_1 \cup \zeta_2 \cup \dots \cup \zeta_o$ alle verbleibende Zyklen - nämlich gar keine - findet. Da Z alle Zahlen der Permutation, also $[1, n]$ enthält, kann auch keine Zahl gefunden werden, die nicht in Z enthalten ist. Somit bricht der Algorithmus mit der leeren Liste ab. Dies ist korrekt, denn es sind bereits alle Zyklen gefunden. Nun gelte, dass `cyclesOf` für ein $x \in \{0, \dots, o\}$ und $Z = \zeta_1 \cup \zeta_2 \cup \dots \cup \zeta_{x-1} \cup \zeta_x$ die Zyklen $\zeta_{x+1}, \dots, \zeta_o$ findet. Wir zeigen, dass dies auch für $x \rightarrow x - 1$ gilt. Zunächst wird der Wert $s \in \{1, \dots, n\}$ ($s = \text{start}$, n ist die Länge von π) mit $s \notin Z$ gesucht. Nun wird der neue Zyklus ζ_x berechnet. Dieser ist sicher disjunkt von den zuvor berechneten Zyklen, da er bei $s \notin Z$ beginnt. Anschließend wird `cyclesOf` rekursiv aufgerufen, mit `handled` = $\zeta_1 \cup \dots \cup \zeta_{x-1} \cup \zeta_x$. Dieser Aufruf liefert nach Induktionsannahme die Zyklen $\zeta_{x+1}, \dots, \zeta_o$. Also werden insgesamt die Zyklen $\zeta_x, \zeta_{x+1}, \dots, \zeta_o$ ausgegeben. Das auch die Sortierung richtig ist, sieht man anhand der Tatsache, dass immer der kleinstmögliche Startwert gesucht wird. Also ist auch dieser Algorithmus korrekt, bei Aufruf von `cyclesOf` mit $Z = \{\}$ werden nämlich die Zyklen ζ_1, \dots, ζ_o zurückgegeben.

Anschließend zeigen wir die Optimalität vom eigentlichem Algorithmus, die Berechnung der Instruktionen. Diese machen wir uns klar, indem wir uns erst für eine beliebige Containerkonstellation, also eine beliebige Permutation überlegen, wie der optimale Weg aussehen muss. Also sei π eine beliebige Permutation über X und $Z := \{\zeta_1, \zeta_2, \dots, \zeta_o\}$ die disjunkten Zyklen, die π darstellen. Nun teilen wir diese Zyklen in die Äquivalenzklassen A_1, \dots, A_a auf. In jeder Äquivalenzklasse sollen - grob gesprochen - nur Zyklen sein, die sich "überschneiden". Wir definieren uns zunächst eine Äquivalenzrelation \equiv Seien $\eta, \theta \in Z$ zwei Zyklen der Länge k bzw. l der Form $e, \eta(e), \dots, \eta^{k-1}(e)$ bzw. $t, \theta(t), \dots, \theta^{l-1}(t)$. Weiter seien $E := \{e, \eta(e), \dots, \eta^{k-1}(e)\}$ und $T := \{t, \theta(t), \dots, \theta^{l-1}(t)\}$, also jeweils die Mengen der "bewegten" Elemente der Zyklen. Sei o.B.d.A. $\min(E) \leq \min(T)$ (Andernfalls vertauschen wir η und θ). Dann gilt $\eta \equiv \theta$ dann, wenn

$\max(T) \leq \max(E)$. Salopp gesagt, gilt $\eta \equiv \theta$ gdw. θ nicht “innerhalb” von η liegt.

Nun machen wir uns noch klar, dass \equiv auch wirklich eine Äquivalenzrelation auf Z ist. Die Reflexivität ist klar, sei $\eta \in Z$ Zyklus und $E :=$ alle Elemente von η (die nicht fest gelassen werden). Dann gilt $\min(E) \leq \min(E)$, also gilt $\eta \equiv \eta$.

Die Symmetrie ist ebenfalls sehr anschaulich, da wir o.B.d.A. $\min(E) \leq \min(T)$ angenommen haben, können wir ebenfalls o.B.d.A. annehmen $\min(T) \leq \min(E)$ und \equiv wäre nach Definition somit symmetrisch.

Also zeigen wir folgend die Transitivität. Seien $\eta, \theta, \iota \in Z, \eta \neq \theta \neq \iota$ und es gelte $\eta \equiv \theta$ und $\theta \equiv \iota$. Seien E, T, I die Mengen der Zyklen η, θ, ι . Dann gilt nach Definition von \equiv $\min(E) \leq \min(T) \leq \min(I) \wedge \max(E) \geq \max(T) \geq \max(I)$, also gilt auch $E \equiv I$. Die Relation \equiv ist also eine Äquivalenzrelation.

Was sagen uns jetzt aber die Äquivalenzklassen? Wir erinnern uns, dass die Zyklen einer Äquivalenzklasse sich “überschneiden”. Das heißt, gibt es zu einer Menge Zyklen mehr als eine Äquivalenzklasse, gibt es wohl Zyklen die sich nicht “überschneiden”.

Betrachtet wir nun nochmals den zunächst gewählten Problemlösungsansatz, nämlich beginnend beim “ersten” Zyklus alle Zyklen zu bearbeiten und bei Überschneidungen zu unterbrechen. Aber wir haben gerade erst gezeigt, dass es eben auch Zyklen gibt, die sich *nicht* Überschneiden! Also muss auch dies (Im Code wäre dies Fall (1) in der Hilfsmethode `step` von `computeCycle`) beachtet werden.

Nun formuliere ich einen Satz über den optimalen Weg von Zyklen.

Der im Sinne der Aufgabenstellung *optimale Weg* w zu einer die Containerpositionen beschreibenden Permutation π mit Länge n , die durch die disjunkten Zyklen Z dargestellt werden kann ist folgender. Sei a die Anzahl der Äquivalenzklassen, in die Z durch \equiv aufgeteilt wird. Dann ist der optimale Weg $w = a \cdot 2 + \sum_{i=1}^n |i - \pi(i)|$.

Dies machen wir uns wie folgt klar. Zunächst betrachte man den Ausdruck $a \cdot 2$. Da *kein* Container über diese “Grenze” gebracht werden muss, aber trotzdem der Kran mindestens einmal zu jeder Position gebracht werden muss, sind hier Leerfahrten nötig. Genauer gesagt sind *zwei* Leerfahrten nötig, da der Kran (mindestens) einmal hinüber und einmal zurück gebracht werden muss. Zurück deswegen, weil er zum Schluss auf jeden Fall an der ersten Position wieder ankommen soll. Folgend setzen wir $w_s := \sum_{i=1}^n |i - \pi(i)|$. Nach meiner Behauptung muss also die Summe der Wege innerhalb der Äquivalenzklassen genau gleich w_s sein. Da immer nur *ein* Container an der Position c auf einmal transportiert werden kann, und zwar jeweils von c nach $\pi(c)$ ist dieser Weg auf jeden Fall zurückzulegen, also muss w auf jeden Fall größer oder gleich w_s sein. Also genügt folgend zu betrachten, dass die Summe w_a der minimalen Wege innerhalb jeder Äquivalenzklasse maximal w_s ist. Angenommen, dies wäre nicht so, also $w_a > w_s$. Dann müsste es neben den Fahrten vom Containerstellplatz zu der dazugehörigen Wagenpositionen noch mindestens eine weitere Fahrt von x nach y geben, auf der kein Container “in die richtige Richtung” transportiert wird. Also entweder kein Container (->Leerfahrt) oder aber ein Container der eigentlich von einem Ort $o \geq x$ zum Wagen $i \leq x$ gebracht werden muss. Da jedoch alle Zyklen innerhalb eine Äquivalenzklasse ohne Leerfahrt abgearbeitet werden können, sind neben den $a \cdot 2$ Leerfahrten zwischen Äquivalenzklassen keine weiteren Leerfahrten nötig. Auch sind keine Fahrten in “falsche” Richtungen innerhalb einer Äquivalenzklasse nötig, da alle Zyklen an einem Stück abgearbeitet werden können. Ein Container i muss zudem nie über die Grenzen einer Äquivalenzklasse transportiert werden, da $\pi(i)$ auf jeden Fall in den selben Grenzen liegt.

Nun muss noch gezeigt werden, dass der Algorithmus alle Voraussetzungen erfüllt und einen optimalen Weg liefert. Dafür müssen lediglich folgende drei Aspekte gezeigt werden.

1. Der erzeugte Weg ist ein zusammenhängender Weg. (Keine Sprünge)

2. Jeder Container wird an die richtige Position gebracht.
3. Es werden keine “unnötige” Fahrten erzeugt.
(Leerfahrten innerhalb einer Äquivalenzklasse, oder Fahrten “in die falsche Richtung”).

Das jeweils die richtige Instruktionen zum Drehen des Krankopfes, Ablegen und Aufnehmen von Containern erzeugt werden, wird hier nicht bewiesen. Es geht hier ausschließlich um den optimalen Weg. Zunächst machen wir uns 1. klar. Sei i die jeweilige Position des Kranes (im Code `prev`). Dann gilt zu Beginn $i = 1$. Hier wird mit der ersten Zyklenabarbeitung begonnen. In jedem Schritt wird zwischen den o.g. 3 Fällen unterschieden. Im ersten Fall gilt $i = \text{max}$ und der nächste Zyklus beginnt bei $\text{max} + 1 = i + 1$. Es wird zunächst die Fahrt `MoveRight` generiert, dann die Zyklen des nächsten Algorithmus angehängt und schließlich wieder eine `MoveLeft` Fahrt generiert. `MoveRight` bewegt den Kran um 1, `MoveLeft` um -1. Also ist der Weg hier zusammenhängend. (Unter der Annahme, dass der Kran durch `cycleInstrs` wieder auf die Ursprungsposition bewegt wird, dies wird unten gezeigt.) Im zweiten Fall gilt $\text{cur} > \text{next}$. Da `nextCycle` bereits abgearbeitet worden wäre, wenn $\text{prev} \geq \text{cur}$, gilt $\text{prev} < \text{cur} < \text{next}$. Der nächste Zyklus überschneidet sich also mit diesem. Auch hier ist der Weg zusammenhängend, wie man sich leicht klar macht, denn nach ausführen von `cycleInstrs` wird der Kran wieder an die Position `next` gebracht. Von dort kann er im nächsten Schritt durch `step` wieder weiter gebracht werden. Im dritten Fall ist es trivial, der Container wird von `prev` nach `cur` gebracht, also ist er ebenfalls zusammenhängend.

Nun zeigen wir die 2. Eigenschaft. Hier unterscheiden wir wieder zwischen den 3 Fällen. Außerdem nehmen wir an, dass wir in einem Schritt immer bereits den vorherigen Container aufgehoben haben, der auf `cur` gebracht werden soll. Im ersten Fall wird der Container erstmal an der Position `max` gelassen, anschließend wird die nächste Äquivalenzklasse abgearbeitet und wieder eins nach links gefahren. Der Kran ist dann wieder an der Position `max` und kann den Container aufnehmen und `step` neu aufrufen, jedoch ohne Nachfolgerzyklen, womit der 3. Fall vorliegt. Im zweiten Fall ist die Überlegung ähnlich. Der aktuelle Container wird bei `next` zwischengespeichert, der Kran fährt den Nachfolgerzyklus ab und kommt wieder an `next` an. Dort nimmt er den zwischengespeicherten Container wieder auf. Es wird im Anschluß wieder `step` aufgerufen, wodurch wieder einer der drei Fälle eintritt. Das nicht immer Fall 2 eintritt, lässt sich daran erkennen, dass jedes mal mindestens ein Zyklus weniger an `step` übergeben wird. Es muss also irgendwann Fall 1 eintreten. Im letzten Fall, dem dritten, wird der aktuelle Container auf `cur` gebracht. Da dies genau die Stelle ist auf der er muss, ist auch die 2. Voraussetzung gegeben. Die 3. Voraussetzung ist ebenfalls gegeben, in obiger Argumentation wurde bereits gezeigt, dass nur Leerfahrten erzeugt werden, die die Äquivalenzklassen verbinden.

Nun müssen lediglich die Annahmen bewiesen werden, von denen ausgegangen wurde. Der obige Beweis ging von der Annahme aus, dass `computeCycle` immer Instruktionen erzeugt, die den Kran wieder auf die Ausgangsposition bringen. Dies soll nun noch gezeigt werden. Da der letzte Container der abgearbeitet wird (innerhalb einem `computeCycle` Aufruf) `first=cycle.head` ist. Da durch die `foldLeft` Anweisungen der Kran zuletzt auf `cur=first` gebracht wird, ist der Kran wieder auf der Ausgangsposition.

Zuletzt soll noch gezeigt werden, dass ein Container nie “auf einen anderen” gelegt wird, sprich dass die “Zwischenspeicherung” funktioniert. Nun, alle Container werden erst dann auf einen Waggon gelegt wenn diese auf ihrer finalen Position sind. Getauscht - wie es in der Aufgabenstellung genannt wird - muss nur im Fall 2, also wenn bei der Abarbeitung des Elements z eines Zyklus ζ bei i ein neuer Zyklus η beginnt und die Abarbeitung unterbrochen werden muss. Zu diesem Zeitpunkt ist der Container $\eta(i)$ an der Position i . Der Kran kann einfach sein bisherigen Container auf die andere Seite schwenken, den Container $\eta(i)$ aufheben und noch-

mals schwenken und den ursprünglichen Container wieder absetzen. Später kommt der Kran wieder zurück und hebt den dort zwischengelagerten Container wieder auf und fährt gemäß einem weiteren `step` Aufruf weiter.

Wir zeigten also, dass der Algorithmus - und damit im Groben auch die Implementierung - optimale Ergebnisse liefert.

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 n traversiert bis ein Wert gefunden wird der noch nicht abgearbeitet - sprich in `handled` enthalten - ist. Nimmt man an, dass das Prüfen auf Enthaltensein konstanten Zeitaufwand darstellt (beispielsweise bei Verwendung eines `HashSet`), 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 `handled` eingefügt. Unter Annahme, dass wiederum 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 $O(c \cdot n)$.

Desweiteren untersuchen wir das Laufzeitverhalten von `computeFromCycles`. Wir wollen beweisen, dass `computeFromCycles` eine Laufzeitkomplexität von $O(n)$ hat. `computeCycle` wird so oft aufgerufen, wie es Container gibt, also c mal. Betrachten wir also die Laufzeitkomplexität eines `computeCycle` Aufrufs abzüglich rekursiver Aufrufe. Innerhalb eines Aufrufs zu einem Zyklus ζ der Länge z wird zunächst der maximal Wert `max` berechnet. Dies hat eine Komplexität von $O(z)$ laut Scala-Dokumentation. Dies liegt nahe, denn der Maximalwert lässt sich durch einmalige Traversierung aller Elemente berechnen. Anschließend erfolgt der `foldLeft`-Ausdruck. Hierbei wird der Zyklus traversiert, also wird `step` $z+1$ mal aufgerufen, da das erste Element noch am Schluss angehängt ist. Welche Komplexität hat nun `step`? Betrachtet man die 3 Fälle genauer, kommt man zu dem Schluss dass `step` eine Komplexität von $O(1)$ hat. Für den Fall 3 ist es trivial. Im Fall 1 wird nix anderes gemacht, als `computeCycle` aufzurufen und eine konstante Menge an Instruktionen anzuhängen. Anschließend wird wieder `step` aufgerufen. Wie oft kommt aber Fall 1 *insgesamt* - also in allen `computeCycle`-Aufrufen zusammen - vor? Da Fall 1 nur bei einer "Äquivalenzklassengrenze" vorkommt und diese danach "auflöst", wird er höchstens a mal ausgeführt, wobei a die Anzahl Äquivalenzklassen ist. Da $a \leq n$, lässt sich die Laufzeit von Fall 1 vernachlässigen, wir wollen ja eine Komplexität von $O(n)$ beweisen. Nun betrachten wir also den Fall 2. Auch in diesem wird nicht mehr als ein `computeCycle`-Aufruf und eine konstante Anzahl Konkatenationen getätigt. Anschließend wird ebenfalls `step` neu aufgerufen. Also betrachten wir wieder, wie oft der Fall 2 eintritt. Er tritt immer genau dann auf, wenn eine Überschneidung von zwei Zyklen vorliegt. Diese wird in Fall 2 danach aufgelöst, wird also nur einmal als Fall 2 bearbeitet. Im Allgemeinen gibt es jedoch bis zu n^2 Überschneidungen, denn es kann sein, dass sich jeder mit jedem schneidet. Daher muss es noch genauer betrachtet werden. Es wird nämlich ein sich überschneidender Zyklus ζ nur einmal als solcher erkannt und abgearbeitet. Das machen wir uns wie folgt klar. Sobald ein Zyklus während der Abarbeitung eines Zyklus η als ein sich überschneidender erkannt wird, wird für diesen ein `computeCycle`-Aufruf getätigt. Dieser liefert neue Zyklen zurück, in denen auf keinen Fall Zyklen sind die sich mit ζ überschneiden, da diese durch den Fall 2 "abgefangen" wurden. Also werden alle Zyklen, die sich mit ζ und η überschneiden nur bei der Abarbeitung von ζ erkannt, nichtmehr jedoch bei folgenden `step` Aufrufen in der Abarbeitung von η . Jeder Zyklus kann also höchstens einmal als ein sich überschneidender Zyklus erkannt werden. Somit fällt auch

der Fall 2 nicht ins Gewicht. Da wir nun gezeigt haben, dass `step` abzüglich der Fälle 1 und 2 (die ja nicht ins Gewicht fallen) eine Komplexität von $O(1)$ hat, ist klar, dass `computeCycle` abzüglich anderer `computeCycle`-Aufrufe eine Laufzeitkomplexität von $O(z)$ hat.

`computeFromCycle` ruft `computeCycle` so auf, dass für jeden Zyklus ein `computeCycle`-Aufruf nötig ist. Jeder Zyklus muss schließlich abgearbeitet werden. Sei im Folgenden Z die disjunkten Zyklen die die Permutation darstellen und $k(\zeta)$ die Länge eines Zyklus ζ . Da für jeden Zyklus $\zeta \in Z$ der Länge z ein Aufruf mit $O(z)$ nötig wird, ist die Gesamtkomplexität

$$O\left(\sum_{i=0}^o O(k(\zeta_i))\right) = O\left(O\left(\sum_{i=0}^o k(\zeta_i)\right)\right) = O(n)$$

Das letzte Gleichheitszeichen gilt, da die Summe der Längen von allen disjunkten Zyklen genau die der Permutation ist.

Somit haben wir die Laufzeit von $O(n)$ für `computeFromCycles` und $O(c \cdot n)$ für alle Algorithmen in der Verkettung gezeigt.

2.3 Ergebnis- und Laufzeitoptimaler 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. Der Speicher hat also einen hohen Speicherplatzverbrauch. In der Praxis heißt dies, dass nur eine Rekursionstiefe von höchstens 10000 möglich ist. Die Entwicklung eines Laufzeitoptimalen Algorithmus betrachte ich als Erweiterung im Sinne der Allgemeinen Hinweise in den Aufgaben. Diese ist sinnvoll, denn - wie später in 4.3 gezeigt - lassen sich damit zu einer Permutation (die die Container darstellt) mit einer Länge in der Größenordnung 10^7 innerhalb weniger Minuten Instruktionen berechnen, die einen optimalen Weg liefern.

Verbesserung Die Verbesserung - und Schwierigkeit - besteht darin, den bisherigen limitierenden Faktor, nämlich die Berechnung der Zyklen zu optimieren. Außerdem müssen alle rekursiven Funktionen umgeschrieben werden, so dass sie vom Scala compiler tail-call optimiert werden können. Das heißt, alle rekursiven Aufrufe einer Funktion müssen der letzte Befehl einer Funktion sein.

Zunächst wurde die Cycle Methode auf folgenden Code optimiert. (Die `@tailrec`-Annotation weist daraufhin, dass die Funktion tail-call optimiert werden kann und soll.)

```

1 def cycle(perm: Seq[Int], start: Int): Cycle = {
2   @tailrec def step(ready: List[Int], idx: Int): Cycle =
3     if(start == idx)
4       ready.reverse
5     else
6       step(idx :: ready, perm(idx - 1))
7   (start :: step(perm(start - 1)))
8 }
```

Die Änderungen betreffen wesentlich die `step` Methode. Diese hat nun zwei Parameter `ready` und `idx`. In `ready` werden alle bisherig gefundene Elemente eines Zyklus akkumuliert. Bei Rekursionsabbruch muss dementsprechend die umgekehrte Liste zurückgegeben werden, da in einer Liste LIFO gilt. Es soll jedoch das zuerst gefundene Element auch als erstes in der Liste stehen. Falls die Abbruchkondition noch nicht erreicht wurde, wird `step` aufgerufen, mit dem aktuellen Index `idx` an `ready` angefügt und dem neuem Index `perm(idx-1)`. Im Gegensatz zum alten `cycle` wird `step` neben dem Startwert `perm(start-1)` zusätzlich noch mit der leeren Liste `Nil` aufgerufen.

Nun wurde auch `cyclesOf` optimiert. Statt in einem `Set` werden die bereits fertigen Zahlen in einem `Boolean-Array` dargestellt. Der Container mit der Nummer `i` ist genau dann bereits abgehandelt, sobald `handled(i-1)=true`. Neben der veränderten Darstellung der fertigen Zahlen werden außerdem zwei zusätzliche Parameter benutzt. Der erste, `ready` speichert ähnlich wie bei dem Neuen `cycle` die vor dem Aufruf "gesammelten" Ergebnisse, in diesem Fall also die bereits gefundenen Zyklen. Der zweite Parameter ist `prev`. Hier wird der Startwert des vorherigen Zyklus - oder wenn es keinen vorherigen gab 0 - übergeben. Die Abbruchbedingung bleibt die gleiche, es wird jedoch die akkumulierten Ergebnisse aus `ready` - wieder wie oben - nach Umkehrung der Reihenfolge zurückgegeben. Die Suche nach dem nächsten Element ist ebenfalls abgeändert. Es wird nichtmehr bei 1 angefangen zu suchen, sondern beim Startwert des vorherigen Zyklus' `prev` um eins nach rechts verschoben. Denn der Startwert des vorherigen Zyklus (und auch alle davor) wurden bestimmt bereits abgearbeitet. Auch die Suchbedingung ist anders, es wird nicht mehr auf nicht-Enthaltsen geprüft, sondern ob im Array an der Stelle `i-1` nicht `true` gesetzt ist. Außerdem müssen die neuen Zahlenwerte nicht mehr in eine Menge eingefügt werden, sondern die Elemente des Arrays an den entsprechenden Indizes müssen auf `true` gesetzt werden. Der rekursive Aufruf erfolgt zudem als letzter Befehl, zusätzlich werden die neuen abgearbeiteten Zyklen `aCycle :: ready` und der Startwert des Zyklus `start` übergeben.

```

1 @tailrec
2 def cyclesOf(ready: List[Cycle], perm: Seq[Int],
3             handled: Array[Boolean], prev: Int): Cycles =
4   (prev+1 to perm.length) find (i => !(handled(i-1))) match {
5     case Some(start) =>
6       val aCycle = cycle(perm, start)
7       for (i <- aCycle)
8         handled(i-1) = true
9       cyclesOf(aCycle :: ready, perm, handled, start)
10    case None =>
11      ready.reverse
12  }
```

Optimale Ergebnisse Wie oben (in 2.2) 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.

Im Prinzip wurden nur mehrere Elemente ersetzt. Die eigentliche Logik gilt immernoch. Insofern liefern auch die neuen Funktionen die gewünschten Zyklen. Die Änderungen wurden oben jeweils so erklärt, dass gleichzeitig die Korrektheit begründet wird.

Optimale Laufzeitkomplexität Die Laufzeit von `cyclesOf` ist $O(n)$, wie ich folgend zeige. Zunächst zeige ich, dass die Gesamtkomplexität aller `cycle` Aufrufe $O(n)$ ist. Jede Zahl wird genau einmal in ein Zyklus eingefügt, da diese disjunkt sind. Weiter zeige ich nun, dass auch die Summe aller anderen Befehle in `cyclesOf` eine Gesamtkomplexität von $O(n)$ aufweisen. Die `for`-Schleife wird nach gleicher Argumentation wie oben ebenfalls insgesamt $O(n)$ -mal durchlaufen. Der Rest sind Operationen, für die nur konstanter Zeitaufwand nötig ist. Die Laufzeitkomplexität hängt nun lediglich von der Suchfunktion ab. Das Prüfen im Array benötigt $O(1)$ Zeit. Da die Suche immer von `prev + 1` bis zum nächsten Wert `start` durchlaufen wird, der später wiederum `prev` im nächsten Aufruf von `cyclesOf` ist, wird auch insgesamt $O(n)$ -mal im Array geprüft. Insgesamt liegt also eine Laufzeitkomplexität von $O(n)$ vor. Da auch `computeFromCycles` $O(n)$ Laufzeitkomplexität vorweisen kann, ist die Gesamtkomplexität $O(n)$.

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**.

Mögliche Parallelisierung Es wurden Überlegungen zur Parallelisierung des Algorithmus zur Berechnung der Instruktionen gemacht. Aus Zeitgründen wurde jedoch auf eine Implementierung verzichtet. Der Algorithmus kann parallelisiert werden, indem zunächst für jeden Zyklus die Instruktionsketten parallel berechnet werden und diese nachträglich kombiniert werden.

3 Implementierung

Die Implementierung gliedert sich folgendermaßen.

Cycler Algorithmen zur Berechnung der Zyklen

(In den zwei Varianten langsamer (`SlowCycler`) und schneller (`FastCycler`))

Instructor Algorithmus zur Berechnung der Instruktionen aus den Zyklen

Gleis Datenstruktur zur Verwaltung der Containerstellplätze und Waggon

Maschine Klasse zur Simulation einer Maschine

ListBuffer modifizierte Variante der standardmäßigem Scala-Klasse `ListBuffer`

Utils hilfreiche Methoden, u.a. zur Ausgabe in Dateien

3.1 Cycler - Berechnung der Zyklen

Da beide Algorithmen zur Zyklenfindung implementiert werden, ist ein `trait Cycler` implementiert, welches die einzige Methode des Moduls `cyclesOf(Seq[Int]): Cycles` definiert. Diese soll zu einer gegebenen Permutation eine Liste von nach Startelementen sortierte Zyklen zurückgeben. Die Implementierung des `SlowCycler` erfolgte wie in 2.2, die des `FastCycler` nach 2.3.

3.2 Instructor - Berechnung der Instruktionen

Anschließend wurden im Modul `Instructor` Funktionen zur Berechnung der Instruktionen erstellt. Diese gliedern sich in die extern zu benutzenden Funktionen sowie die intern benötigten

Hilfsfunktionen. Extern zu verwenden sind `compute(Seq[Int], Cycler): Seq[Instruction]` und `computeFromCycles(Cycles): Seq[Instruction]`. Die letztere berechnet die Liste der Instruktionen aus (meist vorher berechneten) Zyklen, während die erstere die Benutzung dadurch vereinfacht, nur die Permutation angeben zu müssen (die Zyklen werden dann automatisch berechnet). Die interne Hilfsfunktion, die die eigentliche Berechnung definiert ist `computeCycle(Cycle, Cycles): (ListBuffer[Instruction], Cycles)`. Diese gibt zu einem zu bearbeitenden Startzyklus und den restlichen Zyklen eine Liste von Instruktionen und eine Liste von unbearbeiteten Zyklen zurück.

3.3 Gleis - Speichern des Zustand

Die Datenstruktur zum Speichern des aktuellen Status' der Container, Containerstellplätze und Waggons wird in der Klasse `Gleis` implementiert. Ein `Gleis` verwaltet zwei Arrays der Länge n . Das erste Array `con` speichert die jeweilige Containernummer auf dem zugehörigen Containerstellplatz. Das andere Array `wag` speichert die jeweilige Nummer des Container auf einem Waggon. Zu Beginn wird das Array `con` mit der Permutation initialisiert.

Ein `Gleis` definiert Methoden zum Setzen und Aufheben - also Löschen - von Containern an einem bestimmtem Index. Ein `Gleis` stellt `takeCon(Int): Int` zum Löschen auf den Containerstellplätzen bzw. `takeWag(Int): Int` zum Löschen auf den Waggons bereit. Die Methoden `putCon((Int, Int)): Int` und `putWag((Int, Int)): Int` werden zum Setzen von Container auf den Containerstellplätzen bzw. den Waggons bereitgestellt. Außerdem wurde die `toString: String` Methode überschrieben, um eine formatierte Ausgabe zu erhalten. Die oben genannten Methoden werden zur Manipulation der Containernummern zu den jeweiligen Containerstellplätzen, bzw. Waggons verwendet. Die genauere Verwendung wird bei späterer Referenz beschrieben.

3.4 Maschine - Interpretieren der Instruktionen

Um die erzeugten Instruktionen interpretieren zu können, wurde die Klasse `Maschine` geschrieben. Diese stellt eine Methode `interpret` dar, die eine Befehlskette ausführt. Eine `Maschine` nutzt ein `Gleis`, um den Zustand zu speichern. Außerdem wurde die Klasse so gestaltet, dass Unterklassen leicht geschrieben werden können, um beispielsweise eine echte Kransteuerung anzubinden.

3.5 ListBuffer - Erweiterung einer Standardklasse

Um die Befehlsketten effizient erstellen zu können wird eine Datenstruktur benötigt, auf der das Anhängen einer zweiten Befehlskette in konstanter Zeit implementiert werden kann. Anschließend muss sie beginnend bei dem zuerst eingefügtem Element der Einfügereihenfolge folgend in linearer Zeit traversierbar sein. Diese Bedingungen erfüllt - leider - keine Standardklasse aus der Scala Collections API. Deswegen wurde die Klasse `ListBuffer` um das Anhängen eines zweiten `ListBuffers` mit konstantem Zeitaufwand erweitert.

3.6 Utils - Helfende Methoden

Weitere Methoden, die nützlich im Rahmen der Nutzung des Programmes sind, jedoch nicht direkt zur Implementierung der Aufgabelösung dienen, wurden in das Modul `Utils` ausgelagert. Besondere Bedeutung hat die Funktion `randPerm`, die zu einer gegebenen Permutationslänge

eine zufällige Permutation berechnet. Außerdem wurden auch Methoden zum Speichern der Instruktionsketten und Permutationen implementiert.

4 Programmabläufe

4.1 Beispiel aus der Aufgabenstellung

Nachfolgend ist der Ablauf, der sich bei Eingabe des Beispiels aus der Aufgabenstellung ergibt, dargestellt.

Zunächst wird die Permutation erzeugt und in `perm` gespeichert.

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

Anschließend werden die Instruktionen erzeugt und in `instrs` gespeichert.

```
1 scala> val instrs = Instructor compute perm
2 instrs: Seq[de.voodle.tim.bwinf.container.Instruction] =
3   List(TakeCon, MoveRight(1), Rotate, TakeCon, Rotate, PutCon,
4         Rotate, MoveRight(1), Rotate, PutWag,
5         TakeCon, MoveLeft(1), Rotate, PutWag,
6         TakeCon, MoveRight(2), Rotate, PutWag,
7         TakeCon, MoveLeft(3), Rotate, PutWag, TakeCon)
```

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

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

Zuletzt soll die Maschine die Instruktionen interpretieren.

```
1 scala> maschine interpret instrs
2 1 2 3 4;(m=8)
3 4 3 2 1;(l=8)
4 -->      (1)
5   -->      (1)
6   <--      (1)
7   ---->    (2)
8  <-----> (3)
9 res0: de.voodle.tim.bwinf.container.Gleis =
10 Container: - - - -
11 Waggon:    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 erfolgen. Andere Möglichkeiten für einen optimalen Weg wären die nachfolgend dargestellte Abläufe. Es gibt also insgesamt vier verschiedenen Fahrpläne, die für das Beispiel einen optimalen Weg ergeben.

1 1 2 3 4;(m=8)	1 2 3 4;(m=8)	1 2 3 4;(m=8)
2 4 3 2 1;(l=8)	4 3 2 1;(l=8)	4 3 2 1;(l=8)
3 ----> (3)	-----> (3)	----> (2)
4 <-- (1)	<---- (2)	<-- (1)
5 <-- (1)	--> (1)	--> (1)

6	-->	(1)	<--	(1)	-->	(1)
7	<-----	(2)	<--	(1)	<-----	(3)

4.2 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.

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

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

```
1 scala> val instrs = Instructor.compute perm
2 instrs: Seq[de.voodle.tim.bwinf.container.Instruction] =
3   List(TakeCon, MoveRight(3), Rotate, TakeCon, Rotate, PutCon,
4         Rotate, MoveRight(4), Rotate, PutWag,
5         TakeCon, MoveRight(4), Rotate, TakeCon, Rotate, PutCon,
6         Rotate, MoveLeft(0), Rotate, PutWag,
7         TakeCon, MoveRight(5), Rotate, PutWag,
8         TakeCon, MoveRight(1), Rotate, PutWag, ...)
```

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

```
1 scala> val maschine = new Maschine(new Gleis(perm), true)
2 maschine: de.voodle.tim.bwinf.container.Maschine =
3 Container: 20 11 2 8 1 16 10 17 19 14 5 12 9 3 13 15 18 4 7 6
4 Waggon:    - - - - - - - - - - - - - - - - - - - - - -
5
6 scala> maschine.interpret instrs
7 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20; (m=138)
8 20 11  2  8  1 16 10 17 19 14  5 12  9  3 13 15 18  4  7  6; (l=138)
9 ----->                                     (3)
10          ----->                                     (4)
11                  ----->                                     (4)
12                          <                                     (0)
13                              ----->                                     (5)
14                                  ---->                                     (1)
15  <----->                                     (14)
16  ----->                                     (16)
17          <----->                                     (14)
18              ----->                                     (10)
19                      <---                                     (1)
20                          <----->                                     (2)
21                              <----->                                     (4)
22                                  ----->                                     (10)
23                  <----->                                     (12)
24                      ----->                                     (3)
25                          ----->                                     (4)
26          <----->                                     (11)
27  <--->                                     (1)
28  ----->                                     (9)
29          <----->                                     (6)
30  <----->                                     (4)
31 res0: de.voodle.tim.bwinf.container.Gleis =
```

```

32 Container:  - - - - - - - - - - - - - - - - - - - -
33 Waggonns:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

```

Permutationen bis zu einer Länge von 20 können wie gezeigt problemlos in der Konsole angezeigt und dargestellt werden. Durch das gewählte - and die Aufgabenstellung angelehnte - Ausgabeformat können auch die zu fahrende Wege gut in der Konsole dargestellt werden. Die Optimalität des Weges kann leicht nachvollzogen werden. In der ersten Zeile ist die anhand der Permutation ausgerechnete mindestens benötigte Weglänge m ausgegeben. In der zweiten Zeile ist die anhand der Instruktionen berechnete Weglänge l ausgegeben. Wie zu sehen, stimmen diese überein.

4.3 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 ($6,4 \cdot 10^6$) 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 darzustellen. Hierfür wurde eine kleine Scala-Methode geschrieben welche im Modul `Utils` zu finden ist.

```

1 scala> val verified = Utils demonstrate 6400000
2 Time used for computing Cycles: 30093
3 Number of cycles: 18
4 Time used: 110879
5 Time used interpreting: 10639
6 verified: Boolean = true

```

Interessant ist hier die Beobachtung, dass es nur 18 Zyklen gibt, bei einer Permutationslänge von 10^7 . Insgesamt wurden 110879 Millisekunden, also 110 Sekunden bzw. knapp 2 Minuten benötigt, um die Instruktionen zu berechnen. Dies ist ein praktischer Beleg der obenn theoretisch bewiesenen guten Laufzeitkomplexität. Nach der Berechnung der Instruktionen wurden diese testweise interpretiert. Hierfür wurden knapp 11 Sekunden benötigt. Zum Schluss wurde außerdem verifiziert, dass jeder Container auf der richtigen Position ist.

5 Programmnutzung

Die Nutzung des Programms erfolgt primär über eine Scala-Console mit richtig eingestelltem Classpath. Um dies einfach zu erreichen, empfehle Ich Ihnen, im Programmordner `Aufgabe2/dist/` die Konsole des Buildprogramm `sbt` mit `./sbt console` zu starten. Anschließend sollten Sie zunächst alle Klassen und Module aus dem Paket `de.voodle.tim.bwinf.container` importieren. Dies lässt sich beispielsweise wie folgt machen.

```

1 scala>import de.voodle.tim.bwinf.container._
2 import de.voodle.tim.bwinf.container._

```

5.1 Permutationen erzeugen

Permutationen erzeugen Sie entweder durch direkte Eingabe oder Sie lassen eine randomisierte Permutation für eine gegebene Länge erzeugen. Um eine Permutation direkt einzugeben können

Sie einfach die Hilfsfunktionen der Scalabibliothek benutzen. Speichern Sie einfach das Bild der Permutation in einer Seq. Die Permutation aus der Aufgabenstellung geben Sie beispielsweise wie folgt ein.

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

Zufällige Permutationen erzeugen Sie mit der Methode `randPerm` im Modul `Utils` unter Angaben einer Länge. Um eine zufällige Permutation der Länge 4 zu generieren, gehen Sie z.B. wie folgt vor.

```
1 scala>val perm = Utils randPerm 4
2 perm: scala.collection.mutable.IndexedSeq[Int] =
3   WrappedArray(4, 2, 1, 3)
```

5.2 Erzeugen der Instruktionen

Nachdem Sie nun eine Permutation erzeugt haben, können Sie die Methode `compute` des Moduls `Instructor` verwenden, um die Instruktionen zu berechnen.

```
1 scala> val instrs = Instructor compute perm
2 instrs: Seq[de.voodle.tim.bwinf.container.Instruction] = List(Take...
```

Sie können auch - wenn Sie wollen - zunächst die Zyklen berechnen, mit dem schnellerem `FastCycler` oder mit dem langsameren `SlowCycler`. Hierzu rufen Sie einfach die Methode `cyclesOf` auf. Z.B. wie folgt.

```
1 scala> val cycles = FastCycler cyclesOf perm
2 cycles: de.voodle.tim.bwinf.container.Cycler.Cycles =
3   List(List(1, 4), List(2, 3))
4
5 scala> val cycles = SlowCycler cyclesOf perm
6 cycles: de.voodle.tim.bwinf.container.Cycler.Cycles =
7   List(List(1, 4), List(2, 3))
```

Anschließend können Sie die Instruktionen auch direkt aus den Zyklen berechnen. Dafür ist die Methode `computeFromCycles` im Modul `Instructor` da.

```
1 scala> val instrs = Instructor computeFromCycles cycles
2 instrs: Seq[de.voodle.tim.bwinf.container.Instruction] = List(Take...
```

5.3 Simulation der Maschine

Nun haben Sie bereits die Instruktionskette erzeugt. Am einfachsten ist es, eine Maschine zu erzeugen, diese die Instruktionen ausführen zu lassen und anschließend die Ausgabe zu betrachten.

Erzeugen der Maschine:

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

Ausführen der Instruktionen:

```
1 scala>maschine interpret instrs
2 1 2 3 4;(m=8)
3 4 3 2 1;(l=8)
4 -->      (1)
5  -->      (1)
6  <--      (1)
7  ---->    (2)
8  <----- (3)
9 res1: de.voodle.tim.bwinf.container.Gleis =
10 Container:  - - - -
11 Waggon:      1 2 3 4
```

5.4 Zeitmessung

Wenn Sie sich zusätzlich noch die Skalierbarkeit nachvollziehen wollen, fordere ich Sie auf die Funktion `demonstrate` im Modul `Utils` auszuprobieren. Um beispielsweise für 100000 Container Instruktionen ausführen zu lassen und anschließend verifizieren zu lassen, ob auch jeder Container am richtigen Platz angekommen ist, führen Sie folgende Befehle aus.

```
1 scala> val verified = Utils demonstrate 100000
2 Time used for computing Cycles: 707
3 Number of cycles: 12
4 Time used: 852
5 Time used interpreting: 82
6 Verifying results...
7 verified: Boolean = true
```

Bemerkung: Die Ausgaben der Konsole wurden per Hand nachformatiert zwecks besserer Einbettung in den Textfluss.

6 Programmtext

Alle Quelldateien finden sich auf der CD unter Aufgabe2/src/

6.1 Cyclcr

```

1 package de.voodlee.tim.bwinf.container
2 import annotation.tailrec
3 import scala.collection.mutable.tim.ListBuffer // <-- custom ListBuffer
4
5 object Cyclcr {
6   type Cycle = List[Int]
7   type Cycles = List[List[Int]]
8 }
9 import Cyclcr._
10 trait Cyclcr extends Function1[Seq[Int], List[List[Int]]] {
11   def apply(perm: Seq[Int]) = cyclesOf(perm)
12   def cyclesOf(perm: Seq[Int]): Cycles
13 }
14 object SlowCyclcr extends Cyclcr {
15   def cycle(perm: Seq[Int], start: Int): Cycle = {
16     def step(idx: Int): Cycle = // Hilfsfunktion
17       if(start == idx)
18         Nil
19       else
20         idx :: step(perm(idx - 1))
21     start :: step(perm(start-1))
22   }
23   def cyclesOf(perm: Seq[Int]): Cycles = cyclesOf(perm, Set())
24   def cyclesOf(perm: Seq[Int], handled: Set[Int]): List[List[Int]] =
25     (1 to perm.length) find (i => !handled.contains(i)) match {
26       case Some(start) =>
27         val newCycle = cycle(perm, start)
28         val newHandled = handled ++ newCycle
29         newCycle :: cyclesOf(perm, newHandled)
30       case None =>
31         Nil
32     }
33 }
34 object FastCyclcr extends Cyclcr {
35   /** Return the list of disjunct cycles sorted ascending by cycle.head */
36   def cyclesOf(perm: Seq[Int]): Cycles =
37     cyclesOf(Nil, perm, new Array[Boolean](perm.length))
38
39   @tailrec private
40   def cyclesOf(ready: List[Cycle], perm: Seq[Int],
41     handled: Array[Boolean], start: Int = 1): Cycles = { // c *
42     val aCycle = cycle(perm, start) // 0(n_c)
43     for (i <- aCycle) { handled(i-1) = true } // 0(n_c)
44     (start to perm.length) find (i => !(handled(i-1))) match { // 0(i_c)
45       case Some(next) =>
46         cyclesOf(aCycle :: ready, perm, handled, next)
47       case None =>
48         (aCycle :: ready).reverse // 0(1)
49     }
50   }
51   /** Small helper function, finding one cycle. */
52   private def cycle(perm: Seq[Int], start: Int): Cycle = { // 0(n_c)
53     @tailrec def step(ready: ListBuffer[Int], idx: Int): Cycle = // 0(n_c)
54       if(start == idx)
55         ready.toList // 0(1)
56       else
57         step(ready += idx, perm(idx - 1))
58     (start :: step(new ListBuffer[Int], perm(start - 1)))
59   }
60 }

```

6.2 Instructor

```

1 package de.voodle.tim.bwinf.container
2 import scala.annotation.tailrec
3 import scala.collection.mutable.tim.ListBuffer
4 import Cyclor._ // import types.
5
6 object Instructor {
7   def compute(perm: Seq[Int], cyclor: Cyclor = FastCyclor): Seq[Instruction] =
8     computeFromCycles(cyclor cyclesOf perm)
9   def computeFromCycles(cycles: Cycles): Seq[Instruction] =
10     TakeCon :: computeCycle(cycles.head, cycles.tail)._1.toList
11
12   /**
13    * Should be called, after a TakeCon!
14    * When a cycle starts, all the containers in the cycles are supposed to be on the
15    * container side.
16    * Container are always transported on the Container side!
17    */
18   private
19   def computeCycle(cycle: Cycle, other: Cycles): (ListBuffer[Instruction], Cycles) = {
20     val max = cycle.max // 0(n_c)
21
22     type Step = (ListBuffer[Instruction], Cycles, Int)
23     @tailrec def step(instrs: ListBuffer[Instruction], cyclesLeft: Cycles,
24       prev: Int, cur: Int): Step =
25       cyclesLeft.headOption match { // Does another Cycle begins between prev and cur?
26         case Some(nextCycle @ (next :: _)) if prev == max && max+1 == next => // (1)
27           val (cycleInstrs, newCyclesLeft) =
28             computeCycle(cyclesLeft.head, cyclesLeft.tail)
29           val extraInstrs = instrs ++=
30             ListBuffer(PutCon, MoveRight, TakeCon) ++=
31             cycleInstrs ++= ListBuffer(MoveLeft, TakeCon)
32           step(extraInstrs, newCyclesLeft, prev, cur)
33         case Some(nextCycle @ (next :: _)) if next < cur => // (2)
34           val (transInstrs, newCyclesLeft) = computeCycle(nextCycle, cyclesLeft.tail)
35           // Move from prev to nextCycle.head (next)
36           val newInstrs = instrs ++=
37             ListBuffer(Move(prev -> next), Rotate, TakeCon, Rotate, PutCon, Rotate) ++=
38             transInstrs
39           step(newInstrs, newCyclesLeft, next, cur)
40         case _ => // (3)
41           val newInstrs = instrs ++=
42             ListBuffer(Move(prev -> cur), Rotate, PutWag, TakeCon)
43           (newInstrs, cyclesLeft, cur)
44       }
45
46     val erster = cycle.head
47     val initial = (ListBuffer[Instruction](), other, erster)
48     val (instrs, cyclesLeft, last) = (initial /: (cycle.tail :+ erster)) {
49       case ((instrs, cyclesLeft, prev), cur) =>
50         step(instrs, cyclesLeft, prev, cur)
51     }
52     (instrs, cyclesLeft)
53   }
54 }

```

6.3 Gleis

```
1 package de.voodle.tim.bwinf.container
2
3 class Gleis(initCon: Seq[Int]) {
4   val length = initCon.length
5   private val con = Seq(initCon: _*).toArray
6   private val wag = new Array[Int](length)
7
8   private def arrTake(arr: Array[Int])(i: Int): Int = {
9     val res = arr(i-1)
10    arr(i-1) = 0
11    res
12  }
13  private def arrPut(arr: Array[Int])(map: (Int, Int)) = map match {
14    case (i, what) =>
15      require(arr(i-1) == 0, "arr(i-1) at " + i + " must be 0, but is " + arr(i-1))
16      arr(i-1) = what
17  }
18
19  def takeWag(i: Int) = arrTake(wag)(i)
20  def takeCon(i: Int) = arrTake(con)(i)
21  def putWag(map: (Int, Int)) = arrPut(wag)(map)
22  def putCon(map: (Int, Int)) = arrPut(con)(map)
23
24  private def arrString(arr: Array[Int]) = // Only print first 100
25    arr take 100 map (i => if(i == 0) "_" else i.toString) mkString "_"
26  override def toString =
27    "Container:_" + arrString(con) + "\n" +
28    "Waggon:___" + arrString(wag)
29
30  // Immutable Vector copies!
31  def container = Vector(con: _*)
32  def waggon = Vector(wag: _*)
33 }
```


6.4 Maschine

```

1 package de.voodle.tim.bwinf.container
2 import annotation.tailrec
3
4 class Maschine(protected val gleis: Gleis,
5               private val print: Boolean = false) {
6   import Maschine._
7   private val length = gleis.length
8   private val numLength = digits(length)
9   private val space = "␣" * (numLength+1)
10  private val arrow = "-" * (numLength+1)
11
12  private def minLength =
13    gleis.container.zipWithIndex.map { case (v,i) => ((i+1)-v).abs } sum
14
15  def log(str: =>Any) = if(print) println(str) else ()
16
17  def logInts(ints: =>Seq[Int]): String =
18    (for(i <- ints) yield {
19      val diff = numLength - digits(i)
20      "␣" * diff + i
21    }) mkString ("␣")
22
23  def interpret(instrs: Seq[Instruction]): Gleis = {
24    log(logInts(1 to length) + ";(m=" + minLength + ")")
25    log(logInts(gleis.container) + ";(l=" + instrs.map(_.len).sum + ")")
26    interpret(instrs.toList,0,0,1)
27  }
28  // Attach point for further actions (for subclasses)
29  protected def act(instrs: List[Instruction]) {}
30
31  @tailrec private
32  def interpret(instrs: List[Instruction], con: Int, wag: Int, idx: Int): Gleis = {
33    act(instrs)
34    instrs match { // Recursively check
35      case Rotate :: xs =>
36        interpret(xs,wag,con,idx)
37      case TakeCon :: xs =>
38        interpret(xs, gleis takeCon idx, wag, idx)
39      case TakeWag :: xs =>
40        interpret(xs, 0, gleis takeWag idx, idx)
41      case PutCon :: xs =>
42        gleis putCon (idx -> con)
43        interpret(xs, 0, wag, idx)
44      case PutWag :: xs =>
45        gleis putWag (idx -> wag)
46        interpret(xs, con, 0, idx)
47      case MoveRight(len) :: xs =>
48        log(space * (idx-1) + arrow * len + ">" +
49          space * (length-len-idx) + "␣(" + len + ")")
50        interpret(xs, con, wag, idx+len)
51      case MoveLeft(len) :: xs =>
52        log(space * (idx-1-len) + "<" + arrow * len +
53          space * (length-idx) + "␣(" + len + ")")
54        interpret(xs, con, wag, idx-len)
55      case Nil => gleis // Do Nothing
56    }
57  }
58  override def toString = gleis.toString
59 }
60 object Maschine {
61   private def digits(num: Int) = (math.log10(num) + 1).floor.toInt
62 }

```

6.5 Instructions

```

1 package de.voodle.tim.bwinf.container
2
3 sealed trait Instruction {
4   def len: Int = 0
5   def short: String = "" + toString.head
6 }
7 case object PutWag extends Instruction
8 case object PutCon extends Instruction
9 case object Rotate extends Instruction
10 case object TakeWag extends Instruction
11 case object TakeCon extends Instruction
12 sealed trait Move extends Instruction {
13   override def short = (toString filter (_.isUpper)) + "(" + len + ")"
14 }
15 object Move {
16   def apply(len: Int): Move =
17     if(len > 0) MoveRight(len)
18     else MoveLeft(-len)
19   def apply(fromTo: (Int, Int)): Move = fromTo match {
20     case (from, to) => Move(to - from)
21   }
22 }
23 case class MoveLeft(override val len: Int) extends Move
24 object MoveLeft extends MoveLeft(1)
25 case class MoveRight(override val len: Int) extends Move
26 object MoveRight extends MoveRight(1)

```

6.6 Utils

```

1 package de.voodle.tim.bwinf.container
2
3 object Utils {
4   import scala.util.Random
5   import scala.collection.mutable.IndexedSeq
6   def randPerm(n: Int) = {
7     // Make sure we don't convert it to an WrappedArray to often.
8     val a: IndexedSeq[Int] = new Array[Int](n)
9     // Init array // 0(n)
10    for (idx <- 0 until n) a[idx] = idx + 1
11    // randomize array // 0(n)
12    for (i <- n to 2 by -1) {
13      val di = Random.nextInt(i)
14      val swap = a(di)
15      a(di) = a(i-1)
16      a(i-1) = swap
17    }
18    a // return array
19  }
20
21  def demonstrate(n: Int) = {
22    val startTime = System.currentTimeMillis
23    val perm = randPerm(n)
24    val cycles = FastCycler cyclesOf perm
25    println("Time used for computing cycles: " + (System.currentTimeMillis - startTime))
26    println("Number of cycles: " + cycles.length)
27    val instrs = Instructor computeFromCycles cycles
28    val endTime = System.currentTimeMillis
29    println("Time used: " + (endTime - startTime))
30    val gleis = new Gleis(perm)
31    val maschine = new Maschine(gleis)
32    maschine interpret instrs
33    println("Time used interpreting: " + (System.currentTimeMillis - endTime))
34    println("Verifying results...")
35    gleis.waggon.zipWithIndex forall (xy => xy._1 == xy._2 + 1)
36  }
37 }

```

6.7 ListBuffer

```

1 package scala.collection.mutable.tim
2 import scala.collection.{mutable, generic, immutable}
3 import mutable._
4 import generic._
5 import immutable.{List, Nil, ::}
6
7 /** A 'Buffer' implementation back up by a list. It provides constant time
8  * prepend and append. Most other operations are linear.
9  *
10 * @author Tim Taubner
11 * @author Matthias Zenger
12 * @author Martin Odersky
13 * @version 2.8.tim
14 * [...]
15 */
16 @serializable @SerialVersionUID(341963961353583661L)
17 final class ListBuffer[A]
18     extends Buffer[A]
19         with GenericTraversableTemplate[A, ListBuffer]
20         with BufferLike[A, ListBuffer[A]]
21         with Builder[A, List[A]]
22         with SeqForwarder[A]
23 {
24   override def companion: GenericCompanion[ListBuffer] = ListBuffer
25
26   import scala.collection.Traversable
27
28   private var start: List[A] = Nil
29   private var last0: ::[A] = _
30   private var exported: Boolean = false
31   private var len = 0
32
33   protected def underlying: immutable.Seq[A] = start
34
35   /** The current length of the buffer.
36    * This operation takes constant time.
37    */
38   override def length = len
39
40   // Implementations of abstract methods in Buffer
41
42   override def apply(n: Int): A =
43     if (n < 0 || n >= len) throw new IndexOutOfBoundsException(n.toString())
44     else super.apply(n)
45
46   /** Replaces element at index 'n' with the new element
47    * 'newelem'. Takes time linear in the buffer size. (except the
48    * first element, which is updated in constant time).
49    *
50    * @param n the index of the element to replace.
51    * @param x the new element.
52    * @throws Predef.IndexOutOfBoundsException if 'n' is out of bounds.
53    */
54   def update(n: Int, x: A) {
55     try {
56       if (exported) copy()
57       if (n == 0) {
58         val newElem = new ::(x, start.tail);
59         if (last0 eq start) {
60           last0 = newElem
61         }
62         start = newElem
63       } else {
64         var cursor = start
65         var i = 1

```

```

67         while (i < n) {
68             cursor = cursor.tail
69             i += 1
70         }
71         val newElem = new :: (x, cursor.tail.tail)
72         if (last0 eq cursor.tail) {
73             last0 = newElem
74         }
75         cursor.asInstanceOf[::[A]].tl = newElem
76     }
77 } catch {
78     case ex: Exception => throw new IndexOutOfBoundsException(n.toString())
79 }
80 }
81
82 // THIS PART IS NEW (by tim8dev):
83
84 /** Appends a single element to this buffer. This operation takes constant time.
85  *
86  * @param x the element to append.
87  * @return this $coll.
88  */
89 def += (x: A): this.type = {
90     val newLast = new :: (x, Nil)
91     append(newLast, newLast, 1)
92 }
93
94 override def ++=(xs: TraversableOnce[A]): this.type = xs match {
95     case some : ::[A] =>
96         append(some, some.last.asInstanceOf[::[A]], some.length)
97     case buff : ListBuffer[A] =>
98         buff.start match {
99             case some : ::[A] =>
100                 if(buff.exported)
101                     buff.copy()
102                 buff.exported = true
103                 append(some, buff.last0, buff.len)
104             case Nil =>
105                 this
106         }
107     case xs =>
108         super.++=(xs)
109 }
110
111 private def append(x: ::[A], last: ::[A], length: Int): this.type = {
112     if(exported) copy()
113     if(start.isEmpty) {
114         last0 = last
115         start = x
116     } else {
117         val last1 = last0
118         last1.tl = x
119         last0 = last
120     }
121     len += length
122     this
123 }
124
125 // END OF NEW PART (by tim8dev).
126
127 /** Clears the buffer contents.
128  */
129 def clear() {
130     start = Nil
131     exported = false
132     len = 0
133 }
134

```

```

135  /** Prepends a single element to this buffer. This operation takes constant
136  *   time.
137  *
138  *   @param x   the element to prepend.
139  *   @return    this $coll.
140  */
141  def += (x: A): this.type = {
142    if (exported) copy()
143    val newElem = new :: (x, start)
144    if (start.isEmpty) last0 = newElem
145    start = newElem
146    len += 1
147    this
148  }
149
150  /** Inserts new elements at the index 'n'. Opposed to method
151  *   'update', this method will not replace an element with a new
152  *   one. Instead, it will insert a new element at index 'n'.
153  *
154  *   @param n     the index where a new element will be inserted.
155  *   @param iter  the iterable object providing all elements to insert.
156  *   @throws Predef.IndexOutOfBoundsException if 'n' is out of bounds.
157  */
158  def insertAll(n: Int, seq: Traversable[A]) {
159    try {
160      if (exported) copy()
161      var elems = seq.toList.reverse
162      len += elems.length
163      if (n == 0) {
164        while (!elems.isEmpty) {
165          val newElem = new :: (elems.head, start)
166          if (start.isEmpty) last0 = newElem
167          start = newElem
168          elems = elems.tail
169        }
170      } else {
171        var cursor = start
172        var i = 1
173        while (i < n) {
174          cursor = cursor.tail
175          i += 1
176        }
177        while (!elems.isEmpty) {
178          val newElem = new :: (elems.head, cursor.tail)
179          if (cursor.tail.isEmpty) last0 = newElem
180          cursor.asInstanceOf[::[A]].tl = newElem
181          elems = elems.tail
182        }
183      }
184    } catch {
185      case ex: Exception =>
186        throw new IndexOutOfBoundsException(n.toString())
187    }
188  }
189
190  /** Removes a given number of elements on a given index position. May take
191  *   time linear in the buffer size.
192  *
193  *   @param n      the index which refers to the first element to remove.
194  *   @param count  the number of elements to remove.
195  */
196  override def remove(n: Int, count: Int) {
197    if (exported) copy()
198    val n1 = n max 0
199    val count1 = count min (len - n1)
200    var old = start.head
201    if (n1 == 0) {
202      var c = count1

```

```

203     while (c > 0) {
204         start = start.tail
205         c -= 1
206     }
207 } else {
208     var cursor = start
209     var i = 1
210     while (i < n1) {
211         cursor = cursor.tail
212         i += 1
213     }
214     var c = count1
215     while (c > 0) {
216         if (last0 eq cursor.tail) last0 = cursor.asInstanceOf[::[A]]
217         cursor.asInstanceOf[::[A]].tl = cursor.tail.tail
218         c -= 1
219     }
220 }
221 len -= count1
222 }
223
224 // Implementation of abstract method in Builder
225
226 def result: List[A] = toList
227
228 /** Converts this buffer to a list. Takes constant time. The buffer is
229  * copied lazily, the first time it is mutated.
230  */
231 override def toList: List[A] = {
232     exported = !start.isEmpty
233     start
234 }
235
236 // New methods in ListBuffer
237
238 /** Prepends the elements of this buffer to a given list
239  *
240  * @param xs    the list to which elements are prepended
241  */
242 def prependToList(xs: List[A]): List[A] =
243     if (start.isEmpty) xs
244     else { last0.tl = xs; toList }
245
246 // Overrides of methods in Buffer
247
248 /** Removes the element on a given index position. May take time linear in
249  * the buffer size.
250  *
251  * @param n    the index which refers to the element to delete.
252  * @return n    the element that was formerly at position 'n'.
253  * @note       an element must exists at position 'n'.
254  * @throws Predef.IndexOutOfBoundsException if 'n' is out of bounds.
255  */
256 def remove(n: Int): A = {
257     if (n < 0 || n >= len) throw new IndexOutOfBoundsException(n.toString())
258     if (exported) copy()
259     var old = start.head
260     if (n == 0) {
261         start = start.tail
262     } else {
263         var cursor = start
264         var i = 1
265         while (i < n) {
266             cursor = cursor.tail
267             i += 1
268         }
269         old = cursor.tail.head
270         if (last0 eq cursor.tail) last0 = cursor.asInstanceOf[::[A]]

```

```

271     cursor.asInstanceOf[::[A]].tl = cursor.tail.tail
272   }
273   len -= 1
274   old
275 }
276
277 /** Remove a single element from this buffer. May take time linear in the
278  * buffer size.
279  *
280  * @param x the element to remove.
281  * @return this $coll.
282  */
283 override def -= (elem: A): this.type = {
284   if (exported) copy()
285   if (start.isEmpty) {}
286   else if (start.head == elem) {
287     start = start.tail
288     len -= 1
289   } else {
290     var cursor = start
291     while (!cursor.tail.isEmpty && cursor.tail.head != elem) {
292       cursor = cursor.tail
293     }
294     if (!cursor.tail.isEmpty) {
295       val z = cursor.asInstanceOf[::[A]]
296       if (z.tl == last0)
297         last0 = z
298       z.tl = cursor.tail.tail
299       len -= 1
300     }
301   }
302   this
303 }
304
305 override def iterator = new Iterator[A] {
306   var cursor: List[A] = null
307   def hasNext: Boolean = !start.isEmpty && (cursor ne last0)
308   def next(): A =
309     if (!hasNext) {
310       throw new NoSuchElementException("next on empty Iterator")
311     } else {
312       if (cursor eq null) cursor = start else cursor = cursor.tail
313       cursor.head
314     }
315 }
316
317 /** expose the underlying list but do not mark it as exported */
318 override def readOnly: List[A] = start
319
320 // Private methods
321
322 /** Copy contents of this buffer */
323 private def copy() {
324   var cursor = start
325   val limit = last0.tail
326   clear
327   while (cursor ne limit) {
328     this += cursor.head
329     cursor = cursor.tail
330   }
331 }
332
333 override def equals(that: Any): Boolean = that match {
334   case that: ListBuffer[_] => this.readOnly equals that.readOnly
335   case _                    => super.equals(that)
336 }
337
338 /** Returns a clone of this buffer.

```

```
339  *
340  *   @return a <code>ListBuffer</code> with the same elements.
341  */
342  override def clone(): ListBuffer[A] = (new ListBuffer[A]) += this
343
344  /** Defines the prefix of the string representation.
345  *
346  *   @return the string representation of this buffer.
347  */
348  override def stringPrefix: String = "ListBuffer"
349 }
350
351 /** $factoryInfo
352 *   @define Coll ListBuffer
353 *   @define coll list buffer
354 */
355 object ListBuffer extends SeqFactory[ListBuffer] {
356   implicit def canBuildFrom[A]: CanBuildFrom[Coll, A, ListBuffer[A]] =
357     new GenericCanBuildFrom[A]
358   def newBuilder[A]: Builder[A, ListBuffer[A]] = new GrowingBuilder(new ListBuffer[A])
359 }
```