29. Bundeswettbewerb Informatik 2010/2011
2. Runde

Tim Taubner, Verwaltungsnummer 29.0108.01

25. 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. In diesem Dokument erläutere ich meine Lösungen der Aufgaben. Ich halte mich an den Gliederungsvorschlag der 1. Runde. Es sind also die Lösungsideen, Algorithmenentwürfe, Implementierungen, Programmabläufe, Programmnutzungsanleitungen sowie Programmtexte enthalten. Eigenständige Erweiterungen werden erläutert und als solche gekennzeichnet.

Sämtliche Dokumentation und die interaktiv ausführbaren Programme sind auf einer separaten CD mitgeliefert. Wie die erstellten Programme gestartet werden können, ist im Abschnitt A, Allgemeines, erläutert. Alle eingereichten Quelldateien und ausführbare Programmdistributionen wurden alleine von mir - Tim Taubner - erstellt.

Inhaltsübersicht

A. Allgemeines 2

B. Erste bearbeitete Aufgabe: (1) Kisten in Kisten in Kisten 4

C. Zweite bearbeitete Aufgabe: (2) Containerklamüsel 46

A. Allgemeines

1. Persönliche Anmerkungen

Der BWInf und ich Der Bundeswettbewerb Informatik hat mich sehr begeistert. 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 meinen Laptop auf und zeige die Dokumentation zur 1. Aufgabe[1] "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 damit möglich ist, knappen und kompakten Code zu schreiben. Ich bin mir durchaus bewusst, dass Scala noch keine weit verbreitete Sprache ist. Trotzdem hoffe ich, dass der moderne und aussagekräftige Code allgemein für Informatiker gut lesbar ist. Deswegen habe ich auch auf Pseudocode verzichtet, weil er als zusätzliche Sprache und Syntax neben Scala und Mathematik eine zusätzliche Komplexität darstellen würde. Stattdessen formuliere ich alle Algorithmen in Scala-Code und erläutere diesen. Der Scala-Code ist meiner Meinung nach ebenso effektiv wie Pseudo-Code. Lediglich wenige funktionale und objektorientierte Konstrukte müssen dem Leser bekannt sein.[2]

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 von meiner Familie - insbesondere von meinem Vater - und meiner Freundin 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 Quelltexte
dist ausführbare Dateien, sowie zur Ausführung benötigte Bibliotheken

Zusätzlich ist die vorliegende Dokumentation unter

TeX-Doku-Einsendung-29010801-Tim-Taubner.pdf

im Wurzelverzeichnis zu finden. Auch die LaTeX-Quelldateien, mit der diese Dokumentation erzeugt wurde, sind im Verzeichnis "TeX-Doku-Quelldateien" zu finden.

3. Systemvoraussetzungen zur Programmausführung

Um die Programmbeispiele ausführen zu können, sollten folgende Voraussetzungen erfüllt sein:

Prozessor:	mindestens 1 GHz, empfohlen: $\geq$ 2 GHz
Arbeitsspeicher:	mindestens 1024 MB, empfohlen: $\geq$ 2 GB
Grafikkarte:	beliebig
Getestete Betriebsysteme:	Windows 7, Linux (Ubuntu 10.04, Kubuntu 10.10)
Java Runtime:	mindestens Version 1.5 (Java 5), empfohlen: $\geq$ 1.6

[1] siehe Einsendung zur 1. Runde, Einsendungsnr. 108
[2] Beispielsweise sollte dem Leser bekannt sein, wie foldLeft, currying, Monaden, etc. funktionieren.

4. Starten der Kommandozeile

4.1. 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. Das kann durch neues Starten der Kommandozeile gemäß folgender Anleitung erfolgen.

4.2. Unter Windows

Unter Windows startet man die Kommandozeile durch: Start → Ausführen → 'cmd' eingeben → Kommandozeile. Nun kann durch Angabe des Laufwerkbuchstabens des CD-Laufwerks (zumeist "E:") auf das Wurzelverzeichnis der CD gewechselt werden.

4.3. Unter GNOME

Unter gängigen GNOME-Distributionen wie z. B. Ubuntu 8 startet man die Kommandozeile durch: Applikationen → System → Terminal. Die CD wird in Standarddistributionen unter /media/disk o.ä. eingehängt. Durch cd /media/disk wechselt man in das Wurzelverzeichnis der CD.

4.4. Unter KDE

Unter aktuellen KDE-Distributionen wie z. B. Kubuntu 9 startet man die Kommandozeile durch: KDE(KMenu) → Applikationen → System → Terminal. Die CD wird in Standarddistributionen unter /media/disk o.ä. eingehängt. Durch cd /media/disk wechselt man in das Wurzelverzeichnis der CD.

4.5. Unter Mac OS X

Leider steht mir kein Mac zur Nutzung bereit, das Öffnen der Konsole sollte jedoch entweder selbsterklärend oder ähnlich der unter KDE/GNOME sein.

5. Starten der Programme

5.1. Starten der Kommandozeile

Nachdem in einer Konsole im Wurzelverzeichnis die CD geöffnet ist, kann die interaktive Scala-Console (REPL) wie folgt gestartet werden. Durch Eingabe von ./run (run.bat in Windows) in der Konsole öffnet sich die interaktive Scala-Console. Der Classpath wurde bereits richtig eingestellt und die Module beider Aufgaben wurden bereits importiert, so dass die einzelnen Anweisungen in den Aufgaben direkt ausgeführt werden können.

Die Scala-Console kann durch Eingabe von exit wieder geschlossen werden.

5.2. Ausführen der Beispiele

Alle Beispiele können nacheinander ausgeführt werden, indem im Wurzelverzeichnis der CD ./examples (examples.bat unter Windows) in der Konsole eingeben wird. Die Beispiele können auch nach Aufgabe getrennt ausgeführt werden, nämlich durch Eingabe von ./aufgabe1 (aufgabe1.bat unter Windows) bzw. ./aufgabe2 (aufgabe2.bat unter Windows).

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

Inhalt der Aufgabendokumentation

1. Lösungsidee . . . 6
2. Algorithmen und Datenstrukturen . . . 6
2.1. Anmerkung zur Notation . . . 6
2.2. Datenstrukturen . . . 6
2.3. Algorithmen . . . 7
2.4. Brute-force-Algorithmus . . . 7
2.4.1. Lösungsidee . . . 7
2.4.2. Laufzeitverhalten . . . 8
2.4.3. (Un)mögliche Parallelisierung . . . 8
2.4.4. Problem der praktischen Nutzung . . . 8
2.5. Brute-force-Algorithmus nach Aufteilung . . . 9
2.5.1. Lösungsidee . . . 9
2.5.2. Laufzeitverhalten . . . 9
2.5.3. Parallelisierung . . . 9
2.6. Inkrementeller Algorithmus . . . 11
2.6.1. Lösungsidee . . . 11
2.6.2. Online-Algorithmus . . . 11
2.6.3. Strategien . . . 11
2.6.4. Offline-Algorithmus . . . 12
3. Implementierung . . . 13
3.1. Kisten . . . 13
3.2. Kistensatz . . . 13
3.3. Kistenpacker . . . 14
3.3.1. OptimalPacker . . . 14
3.3.2. AufteilenderPacker . . . 14
3.3.3. OnlineAlgo und OfflinePacker . . . 14
4. Programmabläufe . . . 16
4.1. Kleines Beispiel . . . 16
4.2. Algorithmus-Vergleich . . . 19
4.3. Kleine Anzahl Kisten . . . 20
4.4. Mittlere Anzahl Kisten . . . 22
4.5. Große Anzahl Kisten . . . 24
4.6. Riesige Anzahl Kisten . . . 25
4.7. Fazit . . . 26
5. Programmnutzung . . . 27
5.1. Erzeugen der Kisten . . . 27
5.2. Erstellung der Schachtelung . . . 28
5.3. Benchmarking eines Algorithmus' . . . 30
5.3.1. Volumina . . . 30
5.3.2. Benchmarking . . . 30

6. Programmtext 32
6.1. Kisten 33
6.2. Kistensatz 37
6.3. Kistenpacker 39
6.3.1. OptimalPacker 39
6.3.2. AufteilenderPacker 39
6.3.3. OfflinePacker und OnlineAlgo 40
6.3.4. Strategien 41
6.4. Helfer 43
6.4.1. KUtils 43
6.4.2. StandardKisten 44
6.4.3. KistenLogger 44

1. Lösungsidee

Um Frau Y. zu helfen, soll ein Algorithmus entwickelt werden, der zu gebenen Kisten diese so ineinander packt, dass die äußersten Kisten in Summe ein möglichst geringes Volumen vorweisen. Meine Grundidee ist es, die Schachtelung der Kisten als binären Baum aufzufassen.

Optimal wäre es natürlich, wenn die äußeren Kisten zusammen gesehen ein minimales Außenvolumen bieten würden. Dies kann in der Tat erreicht werden, wenn alle möglichen Schachtelungen berechnet werden und dann die mit dem kleinsten Außenvolumen zurückgegeben wird. Um jedoch nicht für jede Kiste nacheinander für jede andere zu überprüfen ob sie hineinpasst, wird eine klare Reihenfolge, in der die Kisten ineinander gepackt werden sollen, gesucht. Dies kann erreicht werden, indem die Kisten dem Volumen nach von groß nach klein sortiert werden. Anschließend werden - beginnend bei der größten - alle Kisten nacheinander zu einer Schachtelung kombiniert. Diesen Ansatz nenne ich Brute-force-Algorithmus und beschreibe ihn in 2.4 genauer. Ohne große Überlegungen lässt sich feststellen, dass solch ein Ansatz prinzipiell immer ein Optimum liefert. Problematisch wird es allerdings, wenn die Laufzeit in Betracht gezogen wird, die zum Berechnen benötigt wird. Denn, wie wir später noch genauer sehen werden, lassen sich bereits Probleme mit kleiner Anzahl Kisten nicht mehr in akzeptabler Zeit lösen. Deswegen betrachten wir nachfolgend noch weitere Ansätze und Algorithmen, die deutlich besser skalieren.

2. Algorithmen und Datenstrukturen

2.1. Anmerkung zur Notation

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

2.2. Datenstrukturen

Um das Problem präzise bearbeiten zu können, möchte ich die von mir verwendeten Datenstrukturen - zum Teil auch mathematisch - beschreiben.

Als einen *Kistenbaum* bezeichne ich einen binären Baum bei dem jeder Knoten eine Kiste repräsentiert. In diesem gilt für jeden Knoten N und die durch ihn repräsentierte Kiste K, dass die Nachfolgerkisten, die durch die null, ein oder zwei Nachfolgerknoten von N repräsentiert werden, gleichzeitig in K passen. Ein Kistenbaum stellt also eine äußere Kiste sowie alle enthaltenen verschachtelten Kisten dar.

Als einen *Kistensatz* bezeichne ich eine Multimenge von Kistenbäumen. Ein Kistensatz kann also eine Lösung im Sinne der Aufgabenstellung darstellen.

Das Grundproblem ist nun, zu gegebenen Kisten $k_1, k_2, \ldots, k_n$ einen Kistensatz ks mit den Wurzeln $w_1, \ldots, w_m$ seiner Kistenbäume zu erzeugen, wobei alle Kisten $k_1, k_2, \ldots, k_n$ genau einmal enthalten sind. Seien $v_1, \ldots, v_m$ die jeweiligen Volumina der Wurzeln $w_1, \ldots, w_m$. Dann gilt es als weitere Aufgabe $\sum_{i=0}^{m} v_i$ zu minimieren.

Die Kisten $k_1, k_2, \ldots, k_n$ werden als Liste aufgefasst, eine Menge ist ungeeignet da Kisten (mit gleichen Maßen) mehrfach vorkommen können.

2.3. Algorithmen

In den nachfolgenden Abschnitten werden insgesamt drei Algorithmen vorgestellt. Der erste im Abschnitt 2.4 liefert zwar optimale Ergebnisse, allerdings ist seine Laufzeitkomplexität unpraktikabel hoch. Es wird deswegen ein Trade-off zwischen Laufzeit und Güte gemacht. Hierzu habe ich zwei weitere Lösungsideen in den Abschnitten 2.5 und 2.6 erstellt, die unterschiedliche Kompromisse darstellen. Es ist offensichtlich, dass Optimalität dadurch nicht erreicht wird. Die grundlegende Motivation ist nun, das Streben nach Optimalität aufzugeben und stattdessen in menschlicher Zeit[1] trotzdem eine gute Packung auch zu einer großen Anzahl von Kisten zu finden. Anders gesagt, wird eine im Allgemeinen suboptimale, aber trotzdem brauchbare und vor allem nach praktischen Maßstäben (Anzahl Kisten, Rechenzeit) sinnvolle Lösung gefunden.

2.4. Brute-force-Algorithmus

2.4.1. Lösungsidee

Der oben bereits skizzierte Algorithmus soll nun präzisiert werden. Dieser Algorithmus `OptimalPacker` hat drei Teile.

1. Die Liste wird entsprechend dem Volumen von groß nach klein sortiert. Das heißt, dass jede Kiste k an der Stelle i, in keine Nachfolgerkiste $k_{i+1}, \ldots, k_n$ passt, weil eine Kiste nur dann in eine andere Kiste passen kann, wenn ihr Volumen kleiner ist. Dies wiederum ermöglicht die Kombination "entlang" der Liste. So müssen andere Reihenfolgen von Kisten nicht berücksichtigt werden, was den Algorithmus erheblich beschleunigt.
2. Die resultierende Liste der Kisten wird nacheinander zu Kistensätzen kombiniert. Die Hilfsfunktion `packSchritt` erzeugt aus einer Menge von Kistensätzen durch Hinzufügen einer gegebenen Kiste die Menge aller möglichen Kistensätze. Diese werden dann weiter mit der nächsten Kiste zu noch mehr Kistensätzen kombiniert. Am Ende sind alle Elemente der Eingabeliste abgearbeitet.
3. Nun wird der Kistensatz ausgewählt, welcher das kleinste Außenvolumen hat und zurückgegeben.

Wichtig ist, um den Code zu verstehen, dass `(satz ++< kiste)` alle Möglichkeiten erzeugt, wie man die Kiste in einen `Kistensatz` einfügen kann.[2]

```
def min(kistenListe: Seq[KisteLeer]) = {
  val kistenSätze = packe(kistenListe)  // Teil 1&2: siehe packe
  kistenSätze min                      //  Teil 3: Minimum (nach Volumen)
   Kistensatz.Ordnung.nachVolumen
}

def packe(kistenListe: Seq[KisteLeer]) = {
  val kisten = sortiere(kistenListe) // Teil 1: Sortieren
  /* Teil 2:
   * Nacheinander für jede Kiste und jeden bereits erzeugten Kistensatz alle
   * möglichen Schachtelungskombiniationen mit dieser Kiste erzeugen. */
  kisten.foldLeft(Set[Kistensatz]()) ( packSchritt )
}

def packSchritt(sätze: Set[Kistensatz], kiste: KisteLeer) =
  if(sätze.isEmpty)
    Set(Kistensatz(kiste :: Nil)) // KistenSatz nur mit der Kiste
  else for { satz <- sätze                // Für jeden satz in sätze
             neuerSatz <- satz ++< kiste // erzeuge neue Möglichkeiten
           } yield neuerSatz            // und gib diese zurück
```

[1] menschliche Zeit << Lebenserwartung eines Menschen (75 Jahre)
[2] Nähere Erläuterungen dazu später, für den Algorithmus ist dies nicht direkt relevant.

2.4.2. 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 $O(n)$. Dies gilt, da jede Kiste k des bestehenden Kistensatzes bis zu n Möglichkeiten bietet, die neue Kiste in k einzupacken. Wir erhalten also $n! + n \cdot (n-1)! + (n-1) \cdot (n-2)! + \cdots + 2 \cdot 1! + 1$. Sprich $O(n \cdot n!)$.

Auch wenn der Worstcase 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 benötigen, unter der Annahme, dass das Prüfen und Hinzufügen einer Kiste in eine andere Kiste eine Nanosekunde dauert.

Unter der gleichen Annahme können etwa 15! Operationen in einer Stunde ausgeführt werden. Unter der Annahme, dass es etwa $x!$ Möglichkeiten für $2x$ Kisten gibt. können $2 \cdot (15-1) = 28$ Kisten in einer Stunde in allen Möglichkeiten gepackt werden.

2.4.3. (Un)mögliche Parallelisierung

Eine Überlegung ist, das Laufzeitverhalten durch Parallelisierung zu verkürzen. Allerdings verspricht dies aufgrund der hohen Laufzeitkomplexität von $O(n \cdot n!)$ kaum Abhilfe. Selbst bei 100 Prozessoren, sprich einer hundertfachen Beschleunigung[3], können gerade mal 17! Operationen pro Stunde ausgeführt werden. Das entspricht $2 \cdot (17-1) = 32$ Kisten. Es können also lediglich etwa 14% $(32/28 = 1,1428\ldots)$ mehr Kisten gepackt werden, was nicht nennenswert ist.

2.4.4. Problem der praktischen Nutzung

Frau Y. hat also ihre mittlerweile 28 Kisten optimal packen können. Dadurch ist nun Platz in ihrem Keller frei geworden. Allerdings empfindet sie es als widersprüchlich, dass sie genau deswegen nicht mehr Kisten in ihren Keller stellen kann, weil sie versucht eine optimale Lösung zu finden, was aber nur für kleine Kistenanzahlen gelingt. Wenn beispielsweise 200 Kisten gepackt werden sollen, bleiben 170 Kisten neben 30 optimal gepackten ungepackt.

[3] Eine solche Beschleunigung wird in der Praxis nie erreicht, es muss immer ein gewisser Overhead für Synchronisation und sequentielle Programmabläufe berücksichtigt werden.

2.5. Brute-force-Algorithmus nach Aufteilung

2.5.1. Lösungsidee

Eine Möglichkeit ist, den Brute-force-Algorithmus immer nur auf eine Gruppe, also einen Teil der Kisten, anzuwenden und danach diese so erzeugten Kistensätze nebeneinander zu stellen.

Sei t die maximale Anzahl Kisten pro Gruppe. Dann funktioniert diese Variante "Brute-force-Algorithmus nach Aufteilung" so, dass der in 2.4 beschriebene Algorithmus für jede Gruppe angewandt wird.

Sei folgend m die Anzahl Gruppen, also die Anzahl der Kisten geteilt durch t. Um die Kisten in Gruppen aufzuteilen, werden die Kisten einmal traversiert, dabei wird die erste Kiste der ersten Gruppe, die zweite Kiste der zweiten Gruppe, usw. und die m. Kiste der m. Gruppe. Dann werden die Gruppen wieder neu gezählt, also wird die $m+1$. Kiste der ersten Gruppe, die $m+2$. der zweiten, usw. zugeordnet.

Nun soll der Algorithmus noch anhand von Scala-Code erläutert und präzisiert werden. Zunächst werden die Kistengruppen mit Hilfe der Funktion `teileKisten` berechnet (Z. 2). Diese Funktion folgt dem oben genanntem Prinzip, ist allerdings sehr technisch und wird daher ausgelassen. Dann wird für jede Kistengruppe der Algorithmus `OptimalPacker` aus 2.4 aufgerufen (Z. 3 - 6). Schließlich werden die Kistensätze nebeneinander gestellt und zurückgegeben (Z. 8 - 10).

```
1  def min(kisten: Seq[KisteLeer]): Kistensatz = {
2    val kistengruppen = teileKisten(kisten)
3    val kistensätze = for { // Für jede Kistengruppe
4      kistengruppe <- kistengruppen
5    } yield
6       OptimalPacker min kistengruppe // packe nach Brute-force
7
8    kistensätze reduceLeft {       // Kombiniere Kistensätze
9      _ neben _                    // stelle diese nebeneinander
10   }
11 }
```

2.5.2. Laufzeitverhalten

Dieser Algorithmus ist bezüglich seiner Laufzeit streng polynomiell. Genauer gesagt, kann er sogar in linearer Zeit ausgeführt werden. Dies ergibt sich aus einer Laufzeitanalyse des Algorithmus.

Sei im Folgenden t der Teilungsfaktor, also die Anzahl Kisten die jeweils eine Gruppe bilden. Das Aufteilen der Kisten in Gruppen braucht $O(n)$. Man erhält also $\frac{n}{t}$ Gruppen. Eine dieser Gruppen zu packen, geschieht in konstanter Zeit. Auch wenn der Brute-force-Algorithmus ursprünglich eine Komplexität von $O(m \cdot m!)$ besitzt, ist mit $m = t$ seine Laufzeit $O(t \cdot t!) = O(1)$, also konstant bei konstantem t. Daraus ergibt sich letztendlich eine Laufzeitkomplexität für Gruppenbildung, Packen und Nebeneinanderstellen von

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

Diese Laufzeitkomplexität ist sogar optimal, denn es muss in jedem Fall jede Kiste einmal betrachtet werden, spätestens bei der Ausgabe der Lösung.

2.5.3. Parallelisierung

Dieser Algorithmus lässt sich schnell und einfach parallelisieren. Die Aufteilung lässt sich parallelisieren, indem $\frac{n}{t}$ parallele Prozesse jeweils immer das t. Element der Liste der Kisten in eine neue anfügen. Diese Prozesse starten jeweils um eins versetzt. Sprich, der erste beim ersten Element, der zweite beim zweiten, usw., der $\frac{n}{t}$. beim $\frac{n}{t}$. Element. Nach der Aufteilung kann jeder Prozess unabhängig von einander den jeweils minimalen Kistensatz berechnen. Anschließend müssen diese zusammengefasst werden. Auch dieser Schritt lässt sich parallelisieren. In jedem Schritt werden die unmittelbar nebeneinander

gestarteten Prozesse zu einem kombiniert, indem die minimalen Kistensätze kombiniert werden. Es sind so also $\log_2(\frac{n}{t})$ Schritte nötig. Der Parallelisierungsgrad nimmt in jedem Schritt um die Hälfte ab.

Nun betrachten wird die benötigte Laufzeit für ein Problem der Größe n auf einem System mit $p := \frac{n}{t}$ Prozessoren. Bis zum Kombinieren der Kistensätze laufen die Prozesse vollständig unabhängig voneinander. Also ist die Zeit hierfür gleich die des größten Einzelproblems, also konstant, wie oben dargestellt. Anschließend fällt der Parallelisierungsgrad mit jedem Kombinationsschritt. Jeder Kombinationsschritt kann in $O(1)$ berechnet werden, da wiederum alle Prozesse parallel arbeiten. (Ein Kombinationsschritt hat konstante Laufzeit.) Es sind jedoch $log_2(\frac{n}{t})$ Schritte notwendig. Die Gesamtlaufzeit ist also

$$O(1) + O(\log_2(\frac{n}{t})) = O(\log(\frac{n}{t}) = O(\log n).$$

Zusammenfassend lässt sich folgendes sagen: *n Kisten lassen sich mit dem Algorithmus "Brute-force nach Aufteilung" auf einem System mit $\frac{n}{t}$ Prozessoren in $O(\log n)$ Zeit zu einem Kistensatz packen.*

Für ein System mit 8 Prozessoren lassen sich bis zu $8 \cdot t = 120$ (mit $t = 15$) Kisten in logarithmischer Zeit packen. Auf modernen Grafikkarten, die mehr als 1000 Prozessoren besitzen, lassen sich also bis zu $1000 \cdot 15 = 15000$ Kisten in logarithmischer Zeit packen.

Leider ist dies nur ein theoretischer Wert. Denn nach dem Amdahlschem Gesetz[4] gibt es immer einen Teil des Algorithmus' der sequentiell ausgeführt werden muss. Außerdem existiert - ebenfalls nach Amdahl[5] - ein gewisser Synchronisationsoverhead.

[4] Goetz, Brian; Peierls, Tim; Bloch, Joshua; Bowbeer, Joseph; Holmes, David; LeaDoug: Java Concurrency in Practice, S. 225 ff., Addison-Wesley, 2006

[5] eben da

2.6. Inkrementeller Algorithmus

2.6.1. Lösungsidee

Ein etwas anderer Ansatz ist, Kistensätze inkrementell zu erzeugen. Das heißt, zu einem - mehr oder weniger gut - gepacktem Kistensatz und einer zu packenden Kiste soll genau ein neuer Kistensatz, der zusätzlich die neue Kiste enthält, geliefert werden. Dieser soll wiederum möglichst gut, also dicht gepackt sein. Eine weitere Beschränkung, die ich an den Algorithmus stelle, ist, dass er in höchstens $O(n)$ Zeit diesen neuen Kistensatz liefert. Mit solch einem Algorithmus lassen sich alle Kisten zusammen in $O(n \cdot n) = O(n^2)$ Zeit packen. Im Folgenden betrachten wir den Algorithmus zunächst als Online-Algorithmus, anschließend betrachten wir mögliche Strategien zum Hinzufügen einer Kiste und schließlich betrachten wir den Algorithmus als Offline-Algorithmus.

2.6.2. Online-Algorithmus

Der oben skizzierte Algorithmus kann so genutzt werden, dass immer sofort die nächste Kiste in einen vorhandenen Kistensatz hinzugefügt wird. Der Algorithmus kann also von Frau Y. verwendet werden, um eine neue Kiste in den bestehenden Kistensatz im Keller hinzuzufügen, ohne alles neu berechnen zu müssen. Der Algorithmus erzeugt einen Kistensatz also unabhängig von später hinzuzufügende Kisten. Es handelt sich also um einen Online-Algorithmus. Die Entwicklung eines Online-Algorithmus' ist in der Aufgabenstellung weder explizit noch implizit gefordert. Dieser stellt also eine Erweiterung dar. Diese fand ich insofern sinnvoll, da sie ein Anwendungsfall direkt erfüllt, nämlich genau den, wenn Frau Y. eine einzelne Kiste erhält. Auf Code wird an dieser Stelle verzichtet. Im Prinzip ist dies wieder ein `foldLeft` über eine Liste, allerdings ist der restliche Code sehr technisch und wird daher ausgelassen.

2.6.3. Strategien

Ein Algorithmus, der eine `Kiste` in linearer Zeit in einen `Kistensatz` packen kann, bezeichne ich als *Strategie*. Es gibt eine Vielzahl an Strategien, recht naheliegend sind unter anderem folgende.

FindeHalbleeren	Sucht eine `KisteHalb`, die noch Platz für die neue `Kiste` bietet.
FindeGroesserenLeeren	Sucht eine `KisteLeer`, die noch Platz für die neue `Kiste` bietet.
FindeZwischenraum	Sucht eine `Kiste`, die durch Entfernen einer Kind-`Kiste` genug Platz für die neue `Kiste` bietet, die wiederum die Kind-`Kiste` aufnehmen kann.
FindeKleinereWurzel	Sucht eine Wurzel-`Kiste`, die in die neue Kiste passt.

Nun sollen die einzelnen Strategien noch genauer erläutert werden. Die Strategien suchen alle in allen Kistenbäumen eines Kistensatz mit Hilfe der Funktion `finde` die erste Kiste, die bestimmte Voraussetzungen erfüllt. Dabei wird jeder Baum traversiert. Treffen die Voraussetzungen auf eine bestimmte `Kiste` nicht zu, so werden alle in dieser `Kiste` gepackten Nachfolgerkisten traversiert. Nachfolgend bezeichne ich mit `kNeu` die einzufügende `Kiste`.

FindeHalbleeren Diese Strategie sucht eine `Kiste kh`, die bereits einen Nachfolger hat und trotzdem Platz für `kNeu` bietet. Nachdem so eine `Kiste` gefunden wird, wird die `Kiste kNeu` in `kh` eingefügt.

FindeGroesserenLeeren Diese Strategie verhält sich ähnlich wie `FindeHalbleeren`. Es wird jedoch eine Kiste `big` gesucht, die leer und größer als `kNeu` ist.

FindeZwischenraum Dies ist die komplizierteste Strategie. Sie sucht eine `Kiste` k, die auf jeden Fall größer ist als `kNeu` und zusätzlich folgende Bedingungen erfüllt. Falls `k` bereits einen Nachfolger `links` hat, so muss `kNeu` größer als `links` sein. Falls `k` jedoch bereits zwei Nachfolger `links` und `rechts` hat, so muss einer der beiden folgenden Fälle erfüllt sein. Entweder der `links` passt in `kNeu` und `k` hat Platz

für `kNeu` und `rechts`, oder `rechts` passt in `kNeu` und `k` hat Platz für `kNeu` und `links`. Natürlich kann es auch sein, dass keiner der Fälle zutrifft. Trifft jedoch ein Fall zu, so packe die neue `Kiste` `kNeu` in den entsprechenden Zwischenraum.

FindeKleinereWurzel Nun wurde noch eine Strategie implementiert, die eine Wurzel `kWurzel` sucht, die in `kNeu` passt. Diese ist recht einfach, sobald gefunden, wird `kWurzel` einfach in `kNeu` gepackt.

Standard-Strategien Diese vier Strategien nenne ich auch *Standard-Strategien*, wenn sie in der Reihenfolge FindeGroesserenLeeren, FindeZwischenraum, FindeHalbleeren, FindeKleinereWurzel verwendet werden.

2.6.4. Offline-Algorithmus

Werden die Kisten vor dem inkrementellem Packen nach Volumen von groß nach klein sortiert, können die Strategien `FindeZwischenraum` und `FindeKleinereWurzel` 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 alle Kisten bekannt sein müssen, handelt es sich nicht mehr um einen Online- sondern einen Offline-Algorithmus. Die Existenzberechtigung dieses Algorithmus' ergibt sich aus der Tatsache, dass bessere Ergebnisse bei vorheriger Sortierung erhalten werden können, als mit dem ursprünglichem Online-Algorithmus.

3. Implementierung

In diesem Abschnitt wird die Implementierung erläutert. Der Programmtext ist in Abschnitt 6 abgedruckt.

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.

3.1. Kisten

Es gibt drei Arten von Kisten: `KisteLeer`, `KisteHalb` und `KisteVoll`. Eine `KisteLeer` enthält keine weitere `Kiste`, eine `KisteHalb` enthält eine `Kiste` und eine `KisteVoll` enthält zwei. Es wurde zunächst ein `trait Kiste` implementiert, welches eine externe Anwendungsschnittstelle bietet und eine interne Implementierungsschnittstelle, welche von den drei Unterklassen implementiert werden muss. Das `trait` wurde als `sealed` implementiert, das heißt, nur Klassen 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 Standardmethoden wie `hashCode: Int` zur Hashcode-Berechnung sowie `equals(Kiste): Kiste` als Definition der Äquivalenz wurden auch zahlreiche anwendungsspezifische Methoden und Felder implementiert. Hier beschränke ich mich auf die Vorstellung der wichtigsten Methoden und Felder.

Eine `Kiste` besitzt die Felder `a,b,c: Int`, die jeweils das größte, zweitgrößte bzw. drittgrößte Außenmaß darstellen. Die Methode `+<(Kiste): Set[Kiste]` liefert als Ergebnis alle Möglichkeiten wie *eine andere* `Kiste` in den durch *diese* `Kiste` definierten Kistenbaum gepackt werden kann. Hierzu wird der gesamte Kistenbaum traversiert, indem die Methode `+<` rekursiv auf allen Nachfolgerkisten aufgerufen wird. Weiter wurden Methoden zum Vergleich zweier Kisten implementiert. Diese sind `>~(Kiste):Boolean` sowie `>=~(Kiste):Boolean`. Die erstere liefert zu einer anderen `Kiste der`, ob diese hineinpasst. Die zweite prüft indes, ob die `Kiste der` kleinere oder gleichgroße Maße hat, also ob `der.a <= a` und `der.b <= b` und `der.c <= c` gilt.

3.2. Kistensatz

Da eine `Kiste` die Wurzel eines Kistenbaums ist und diesen repräsentiert, (betrachten Sie z. B. eine `KisteLeer` als einen einelementigen Kistenbaum), muss ein Kistensatz lediglich Referenzen auf die einzelnen `Kiste`-Objekte speichern. Es ergeben sich für die Datenstruktur, die den Kistensatz verwaltet, folgende drei Voraussetzungen.

1. Das Ersetzen eines [Teil]baums 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 eine neues Objekt zurück. Dieses Objekt muss dann durch eine Lösch- und eine Hinzufügeoperation in den Baum des `Kistensatz` aktualisiert werden.
2. Duplikate müssen zugelassen sein, da es passieren kann, das zwei Kisten genau die gleichen Maße haben.
3. Die Kistenbäume eines Kistensatzes müssen so sortiert sein, dass ein Vergleichen nach Elementen zwischen zwei Kistensätzen 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 `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 das Hinzufügen einer Kiste k (Scala: `+(Kiste):Kistensatz`) mit dem Hinzufügen von $k \to 1$ in den Binärbaum, bzw. wenn k schon erhalten ist, mit dem Setzen von $k \to i_k + 1$.

Analog erfolgt auch das Entfernen einer Kiste k (Scala: `-(Kiste):Kistensatz`), daher wenn k nicht enthalten oder $i_k = 1$ entferne k aus dem Binärbaum, andernfalls setze $k \to i_k - 1$.

Außerdem wurde noch die Methode `+<(Kiste): Set[Kistensatz]` implementiert, die zu einer Kiste die Menge aller Kistensätze liefert, in denen die neue Kiste in einer alten Kiste verpackt ist. Weiter liefert `++<(Kiste): Set[Kistensatz]` zusätzlich zu den durch `+<` berechneten Kistensätzen den durch `+` berechneten `Kistensatz`. Das heißt, `++<` liefert alle Kistensätze, die durch Hinzufügen der Kiste an einer beliebigen Stelle - also entweder in einer alten `Kiste` oder neben dem `Kistensatz` - möglich sind.

3.3. Kistenpacker

Da verschiedene Algorithmen entwickelt wurden, bietet es sich an, von mehreren Algorithmen verwendete Funktionen auszulagern. Dies erfolgte in Scala durch Verwendung einer Klassenhierarchie. Die Wurzel der Hierarchie ist das `trait Kistenpacker`. Dieses definiert das Feld `kisten: List[KisteLeer]` sowie die Methode `min: Kistensatz`. `kisten` stellt die Liste der Kisten dar, die abgearbeitet werden sollen. `kisten` muss jedoch nicht die Eingabeliste sein, es ist beispielsweise auch möglich, dass eine Unterklasse die Kisten sortiert bevor sie abgearbeitet werden. Die Methode `min` soll den Kistensatz liefern, der möglichst geringes Außenvolumen aufweist.

Zusätzlich wurde die Klasse `SortierenderPacker` geschrieben, die das Feld `kisten` von `Kistenpacker` überschreibt und diese mit den Kisten der sortierten Eingabeliste `kistenListe` initialisiert.

3.3.1. OptimalPacker

Der Brute-force-Algorithmus wurde in dem `object OptimalPacker` implementiert. Dieses wurde in etwa mit dem bereits im Entwurf dargestellten Code implementiert. Die Methode `min`, die das Minimum zurück liefern soll, ruft `packe` auf, nachdem die Eingabeliste sortiert wurde. Dann wird aus der so berechneten Menge aller Kistensätze das Minimum ausgewählt, wobei die Ordnung nach Volumen benutzt wird.

3.3.2. AufteilenderPacker

Der `Kistenpacker AufteilenderPacker` implementiert lediglich die Methode `min`. In dieser wird zunächst der Wert `m` berechnet, der die Anzahl der Gruppen darstellt. Dann wird das Feld `geteilteKisten` berechnet. Dieses soll die Gruppen als `Vector` von `List`en darstellen. Beginnend bei einem `Vector` mit leeren `List`en wird ein `foldLeft` über die `kisten` durchgeführt. Es werden die `Kisten` in jedem Schritt an die im `Vector` nächste `List` angefügt. Beim Erreichen des Endes vom `Vector` wird wieder vom Anfang des `Vector`s angefangen. Anschließend wird für jede Gruppe das Minimum berechnet und anschließend durch einen `reduceLeft` Aufruf zu einem `Kistensatz` reduziert.

3.3.3. OnlineAlgo und OfflinePacker

Um den `OfflinePacker` gut implementieren zu können, wurde zunächst die Klasse `OnlineAlgo` implementiert, die den aktuellen `Kistensatz` und eine Liste von Strategien enthält. Die Klasse `OnlineAlgo` implementiert zwei Methoden. Die eine Methode `+(KisteLeer): OnlineAlgo` liefert den `OnlineAlgo` mit dem `Kistensatz`, der durch Anwendung der Strategien die neue `Kiste` enthält. Hierbei werden alle Strategien mit `foldLeft` durchlaufen, wobei so lange versucht wird eine Strategie anzuwenden, bis eine passende gefunden wurde. Wurde keine passende gefunden, wird die `Kiste` einfach neben den `Kistensatz` gestellt. Die andere Methode `++(Seq[KisteLeer]): OnlineAlgo` liefert einfach zu einer Sequenz von Kisten durch nacheinander Aufrufen von `+` mit einem `foldLeft`-Aufruf einen `OnlineAlgo` mit einem `Kistensatz` der alle neuen Kisten enthält.

Mit dieser Klasse gestaltet sich die Implementierung des `OfflinePacker`s sehr einfach. Die Methode `min` muss lediglich die Methode `++` eines leeren `OnlineAlgo` aufrufen und dann den `Kistensatz`, der sich dadurch ergibt, zurückgeben.

Strategien Nun sollen noch die Implementierung der Strategien erläutert werden. Diese folgt den Prinzipien, die in 2.6.3 beschrieben sind. Es ist bei allen Strategien außer `FindeKleinereWurzel` jedoch noch nötig, den gesamten Kistenbaum "über" der gefunden Kiste anzupassen. Da alle Datenstrukturen unveränderbar sind, muss entlang des Pfades nach oben der Kistenbaum neu aufgebaut werden. Deswegen liefert die Methode `find` von `Kistensatz`, mit der die Kiste gesucht wird, einen Pfad bis zur neuen Kiste zurück. Entlang diesem kann dann der Kistenbaum neu aufgebaut werden.

4. Programmabläufe

Nachfolgend sind mehrere Programmabläufe der verschiedenen Algorithmen abgebildet. Soweit möglich, werden jeweils alle Algorithmen mit der gleichen Eingabeliste von Kisten ausgeführt.

4.1. Kleines Beispiel

Betrachten wir folgende 7 Kisten und deren Gesamtvolumen:

```
scala> val kisten = ( 6 x  6 x  2) :: (6 x 4 x 2) :: (6 x 6 x 4) ::
                    ( 8 x  8 x  8) :: (8 x 8 x 6) :: (6 x 4 x 2) ::
                    (10 x 10 x 10) :: Nil
kisten: List[de.voodle.tim.bwinf.kisten.KisteLeer] =
  List(KisteLeer(6,6,2), KisteLeer(6,4,2), KisteLeer(6,6,4),
       KisteLeer(8,8,8), KisteLeer(8,8,6), KisteLeer(6,4,2),
       KisteLeer(10,10,10))

scala> val v = kisten.map(_.v).sum // Das Gesamtvolumen der Kisten
v: Int = 2208
```

OptimalPacker

Als Erstes werden die Kisten mit `OptimalPacker` gepackt. Es werden zunächst alle Möglichkeiten berechnet und anschließend nur das Minimum (vgl. Z. 31).

```
// Alle Möglichkeiten erzeugen
scala> val optKss = OptimalPacker packe kisten
optKss:
 scala.collection.immutable.Set[de.voodle.tim.bwinf.kisten.Kistensatz]=
Set({
KisteHalb(8,8,8)
 \-KisteLeer(6,6,4),
KisteVoll(10,10,10)
 \-KisteVoll(8,8,6)
  \-KisteLeer(6,6,2)
  \-KisteLeer(6,4,2)
 \-KisteLeer(6,4,2)
}, {
KisteLeer(6,4,2),
KisteLeer(6,6,2),
KisteLeer(6,6,4),
KisteLeer(8,8,8),
KisteVoll(10,10,10)
 \-KisteLeer(8,8,6)
 \-KisteLeer(6,4,2)
}, {
KisteLeer(6,4,2),
KisteLeer(6,6,4),
KisteLeer(8,8,6),
KisteLeer(8,8,8),
KisteVoll(10,10,10)
 \-KisteLeer(6,6,2)
 \-KisteLeer(6,4,2)
}, { ...
```

```
scala> val anzahl = optKss.size // Anzahl berechneter Kistensätze
anzahl: Int = 216

scala> val opt = OptimalPacker min kisten
opt: de.voodle.tim.bwinf.kisten.Kistensatz =
{
KisteVoll(8,8,6)
 \-KisteLeer(6,4,2)
 \-KisteLeer(6,4,2),
KisteHalb(10,10,10)
 \-KisteVoll(8,8,8)
  \-KisteLeer(6,6,4)
  \-KisteLeer(6,6,2)
}

scala> val v = opt.v
v: Int = 1384
```

AufteilenderPacker

Als Zweites wird der Algorithmus `AufteilenderPacker` genutzt. Der Aufteilungsgrad sei $t = 4$. Die Kisten werden also wie folgt in die vier Gruppen aufgeteilt.

1. und 5. Kiste	in die 1. Gruppe,
2. und 6. Kiste	in die 2. Gruppe,
3. und 7. Kiste	in die 3. Gruppe,
sowie 4. Kiste	in die 4. Gruppe.

```
scala> val auf = AufteilenderPacker(4) min kisten
auf: de.voodle.tim.bwinf.kisten.Kistensatz =
{
KisteVoll(8,8,8)
 \-KisteLeer(6,4,2)
 \-KisteLeer(6,4,2),
KisteVoll(10,10,10)
 \-KisteHalb(8,8,6)
  \-KisteLeer(6,6,4)
 \-KisteLeer(6,6,2)
}

scala> val v = auf.v
v: Int = 1512
```

Wir sehen hier, dass das Volumen größer ist als beim Optimum. Dies liegt daran, dass in diesem Beispiel die Kisten `(10 x 10 x 10)` und `(8 x 8 x 6)` in der gleichen Kistengruppen sind. Diese müssten jedoch beide Wurzel sein, um eine optimale Packung erreichen zu können.

OfflinePacker

Schließlich wird mit `OfflinePacker` gepackt. Es wird ein `OfflinePacker` mit leerer Parameterliste erzeugt, somit werden die `standardStrategien` benutzt und die Kisten vor dem Packen sortiert.

```
scala> val off = OfflinePacker() min kisten
off: de.voodle.tim.bwinf.kisten.Kistensatz =
{
KisteLeer(6,4,2),
KisteHalb(8,8,6)
 \-KisteLeer(6,6,4),
KisteHalb(10,10,10)
 \-KisteVoll(8,8,8)
  \-KisteLeer(6,6,2)
  \-KisteLeer(6,4,2)
}

scala> off.v
res2: Int = 1432
```

Hier ist das Volumen ebenfalls größer als beim Optimum `opt`, jedoch kleiner als `auf`. Die Strategien des `OfflinePacker` finden für die letzte zu packende Kiste (6 x 4 x 2) keinen Platz mehr.

4.2. Algorithmus-Vergleich

In diesem Abschnitt sollen nun die Algorithmen für größere Beispielen miteinander verglichen werden. Wir betrachten insbesondere die Packdichte und die Laufzeit. Wir vergleichen bei den kleinen Beispielen die Algorithmen Brute-force, Brute-force nach Aufteilung, sowie OfflinePacker. Bei den mittleren und größeren Beispielen müssen wir auf den Algorithmus Brute-force verzichten, denn dieser lässt sich nicht mehr in passabler Zeit ausführen.

Hinweis Alle Zeitangaben sind in Millisekunden. Das Verhältnis zwischen der Summe aller Volumina der ungepackten Kisten zu der Summe der Endvolumina der äußeren Kisten bezeichne ich als *Komprimierungsgrad*. Es gilt also

$$\text{Komprimierungsgrad} = \frac{\sum \text{Volumina der ungepackten Kisten}}{\sum \text{Envolumina der äußeren Kisten}}.$$

Hilfsfunktion Zunächst wird eine Hilfsfunktion definiert, die zu einer Anzahl eine Funktion zurück liefert, die entsprechend viele Kisten zufällig erzeugt. Die Kisten haben dabei maximal eine Abmessung von 80 x 80 x 80.

```
scala> def kisten(anzahl: Int) =
  () => StandardKisten.zufallKisten(80,80,80) take anzahl toList
kisten: (anzahl: Int)() => List[de.voodle.tim.bwinf.kisten.KisteLeer]
```

Außerdem definieren wir uns noch eine Funktion `großeKisten`, die ähnlich wie die obere Funktion zufällig erzeugte Kisten zurück gibt. Allerdings erzeugt diese Funktion Kisten mit Abmessungen bis zu 1000 x 1000 x 1000.

```
scala> def großeKisten(anzahl: Int) =
  () => StandardKisten.zufallKisten(1000,1000,1000) take anzahl toList
großeKisten:(anzahl: Int)()=>List[de.voodle.tim.bwinf.kisten.KisteLeer]
```

4.3. Kleine Anzahl Kisten

Unter einer kleinen Anzahl Kisten verstehe ich etwa 10 Kisten. Nachfolgend wird also immer mit 8, 10 bzw. 12 Kisten gemessen. Es werden nun nacheinander verschieden Algorithmen - bzw. Varianten davon - auf Packdichte und Laufzeit analysiert. Nachfolgend wird jeweils für jeden Algorithmus eine Musterausführung dargestellt. Anschließend werden die gewonnenen Ergebnisse in einer Tabelle zusammengefasst gegenübergestellt.

OptimalPacker Der Algorithmus `OptimalPacker` wird 1000-mal nacheinander mit 8, 10, bzw. 12 Kisten ausgeführt. Aufgrund der schlechten Laufzeitkomplexität, werden bei 12 Kisten jedoch nur 100 Abläufe ausgeführt. Die folgenden Zeilen 1-7 zeigen das Beispiel für 8 Kisten.

```
scala> val (optKd, optZeit) =
  KUtils.bench(OptimalPacker, kisten(8), 1000, verbose = false)
Packer: OptimalPacker
Wiederholungen  Komprimierungsgrad      Zeit
8               1,4                     3
optKd: BigDecimal = 1.4043923926972448
optZeit: Long = 3

[...] // Aufrufe mit 10 bzw. 12 Kisten
```

AufteilenderPacker Den Aufteilungsgrad t setze ich zunächst immer auf die Hälfte der Anzahl von Kisten, also 4, 6 bzw. 8. Anschließend teste ich nochmal mit Kistenanzahlen von 10 und 12, jedoch mit gleichbleibendem Aufteilungsgrad $t = 4$. Gezeigt ist hier ebenfalls das Beispiel für 8 Kisten.

```
scala> val (aufKd, aufZeit) =
  KUtils.bench(AufteilenderPacker(4), kisten(8), 1000, verbose = false)
Packer: AufteilenderPacker(4)
Wiederholungen  Komprimierungsgrad      Zeit
1000            1,23                    0
aufKd: BigDecimal = 1.228
aufZeit: Long = 0

[...] // Aufrufe mit 10 bzw. 12 Kisten
```

OfflinePacker Schließlich wird auf gleiche Weise der `OfflinePacker` mit den Standard-Strategien getestet. Gezeigt ist wieder der Fall für 8 Kisten.

```
scala> val (offKd, offZeit) =
  KUtils.bench(OfflinePacker(), kisten(8), 1000, verbose = false)
Packer: OfflinePacker(OnlineAlgo({

},List(FindeGroesserenLeeren, FindeZwischenraum,
       FindeHalbleeren, FindeKleinereWurzel)),true)
Wiederholungen  Komprimierungsgrad      Zeit
1000            1,32                    0
offKd: BigDecimal = 1.325
offZeit: Long = 0

[...] // Aufrufe mit 10 bzw. 12 Kisten
```

In den beiden folgenden Tabellen sind alle Ergebnisse der Beispielprogrammausführungen zusammengefasst.

Komprimierungsgrad

Algorithmus	**8 Kisten**	**10 Kisten**	**12 Kisten**
OptimalPacker	1,40	1,46	1,54
AufteilenderPacker ($t = \frac{n}{2}$)	1,23	1,27	1,32
AufteilenderPacker ($t = 4$)	1,22	1,19	1,22
OfflinePacker	1,32	1,39	1,43

Den höchsten Komprimierungsgrad erzielt wie erwartet der `OptimalPacker`. Der `AufteilenderPacker` fällt deutlich hinter den `OfflinePacker` zurück. Außerdem ist der Komprimierungsgrad bei $t = \frac{n}{2}$ ein Stück weit größer als bei $t = 4$. Auffallend ist außerdem, dass bei 8 Kisten beide Varianten des Algorithmus' `AufteilenderPacker` genau die gleichen Parameter übergeben bekommen (denn $t = \frac{8}{2} = 4$) und trotzdem abweichende Komprimierungsgrade erreichen. Dies liegt daran, dass die Kisten für jeden Aufruf neu zufällig erzeugt werden und somit kleine Abweichungen unvermeidlich sind.

Laufzeit [in Millisekunden]

Algorithmus	**8 Kisten**	**10 Kisten**	**12 Kisten**
OptimalPacker	3	49	1273
AufteilenderPacker ($t = \frac{n}{2}$)	0	0	0
AufteilenderPacker ($t = 4$)	0	0	0
OfflinePacker	0	0	0

Bereits hier kann erkannt werden, wie schlecht der Brute-force-Algorithmus skaliert. Bei lediglich 2 Kisten mehr braucht `OptimalPacker` bereits bis zu 25 mal ($49 * 25 = 1225 < 1273$) länger.

Es lässt sich ebenfalls erkennen, dass sowohl `OfflinePacker` als auch `AufteilenderPacker` - gegenüber dem `OptimalPacker` - bei kleinen Kistenzahlen wesentlich schneller sind, so dass die Laufzeiten (bei diesen kleinen Kistenanzahlen) nicht einmal gemessen werden können und somit zu 0 gerundet werden.

4.4. Mittlere Anzahl Kisten

Nach dem Ausscheiden von `OptimalPacker` testen wir nun in der Größenordnung von 100 Kisten. Genauer gesagt, testen wir mit den Anzahlen 88, 100 sowie 128. Allerdings testen wir nun nicht mehr mit 1000 Wiederholungen, sondern mit 256 Wiederholungen, um zügig testen zu können. Um zu sehen, wie sich die Algorithmen bei unterschiedlichen Größen der Kisten verhalten, wird zusätzlich mit der Funktion `großeKisten` getestet. Beim aufteilenden Packer ist natürlich wieder die Frage, welcher Teilungsfaktor sinnvoll ist. Er sollte auf jeden Fall nicht größer als 10 sein, da die Laufzeit exponentiell mit t steigt und größere Werte hier leider nicht praktikabel sind. Ich entschied mich für 8 und 10 als Teilungsfaktor.

In den beiden folgenden Tabellen sind alle Ergebnisse der Beispielprogrammausführungen zusammengefasst.

Komprimierungsgrad

Algorithmus	**88 Kisten**	**100 Kisten**	**128 Kisten**
Kleine Kisten			
AufteilenderPacker ($t = 8$)	1,40	1,39	1,40
AufteilenderPacker ($t = 10$)	1,47	1,47	1,47
OfflinePacker	1,96	2,00	2,07
OfflinePacker (eine Strategie)	1,24	1,25	1,25
Große Kisten			
AufteilenderPacker ($t = 8$)	1,41	1,40	1,41
AufteilenderPacker ($t = 10$)	1,46	1,46	1,44
OfflinePacker	2,49	2,57	2,72
OfflinePacker (eine Strategie)	2,27	2,35	2,48

Der erreichte Komprimierungsgrad mit `OfflinePacker` ist größer als 2 und damit deutlich über dem Komprimierungsgrad, der mit `AufteilenderPacker` erreicht werden konnte. Außerdem lässt sich beobachten, dass der Komprimierungsgrad mit steigender Kistenzahl steigt. Dies lässt sich damit erklären, dass die Kistensätze durch mehr Kisten, besser "gesättigt" werden können. Interessanterweise liefert der aufteilende Packer auch mit $t = 10$ kaum bessere Werte als mit $t = 8$. Allerdings braucht er - wie wir unten noch genauer betrachten - deutlich länger, da die Laufzeit exponentiell mit t steigt. Als Vergleich werden hier noch die Ergebnisse des einfachsten `OfflinePacker`s mit der einzigen Strategie `FindeGroesserenLeeren` und ohne Sortieren vor dem Packen angegeben.

Laufzeit [in Millisekunden]

Algorithmus	**88 Kisten**	**100 Kisten**	**128 Kisten**
Kleine Kisten			
AufteilenderPacker ($t = 8$)	33	30	49
AufteilenderPacker ($t = 10$)	375	490	792
OfflinePacker	5	7	13
OfflinePacker (eine Strategie)	3	4	7
Große Kisten			
AufteilenderPacker ($t = 8$)	41	37	59
AufteilenderPacker ($t = 10$)	477	680	732
OfflinePacker	3	4	7
OfflinePacker (eine Strategie)	2	3	5

Über die Laufzeit lassen sich hier nur Vermutungen anstellen, da die Werte absolut gesehen klein bleiben. Beim Algorithmus `AufteilenderPacker` ist der sprunghafte Anstieg der Laufzeit bei leicht steigendem

t zu beobachten. Dies kommt durch die exponentielle Laufzeit des `OptimalPacker`, welcher vom aufteilenden Packer benutzt wird. Absolut gesehen lässt sich noch die niedrigere Laufzeit des `OfflinePacker` erkennen. Als vernünftige Indizen für die Laufzeitkomplexität sind die Ergebnisse nicht geeignet, da die Abweichungen bei diesen kleineren Beispielen relativ gesehen noch zu hoch sind.

4.5. Große Anzahl Kisten

Nun sollen große Anzahlen von Kisten getestet werden. Genauer gesagt 800, 1600 und 3200 Kisten. Neben dem Variieren der Kistenanzahl, werden wir außerdem noch die Größe der Kisten verändern. Um die absolute Laufzeit niedrig zu halten, werden außerdem nur noch 64 Wiederholungen ausgeführt. Außerdem teste ich weiterhin mit den Teilungsfaktoren 8 bzw. 10.

Komprimierungsgrad

Algorithmus	**800 Kisten**	**1600 Kisten**	**3200 Kisten**
Kleine Kisten			
AufteilenderPacker ($t = 8$)	1,40	1,40	1,40
AufteilenderPacker ($t = 10$)	1,46	1,47	1,47
OfflinePacker	2,80	3,12	3,47
OfflinePacker (eine Strategie)	1,28	1,29	1,30
Große Kisten			
AufteilenderPacker ($t = 8$)	1,41	1,41	1,41
AufteilenderPacker ($t = 10$)	1,46	1,47	1,47
OfflinePacker	4,45	5,49	6,64
OfflinePacker (eine Strategie)	4,01	4,80	5,80

Der mit `OfflinePacker` erreichte Komprimierungsgrad ist jetzt sogar deutlich über doppelt so groß wie der Komprimierungsgrad des aufteilenden Packers. Dies lässt sich im direktem Vergleich erkennen. Während der Komprimierungsgrad bei `AufteilenderPacker` konstant schwach auf etwa 1,4 bleibt, steigt der Komprimierungsgrad beim `OfflinePacker` bis auf über 6 an. Dies lässt sich an der Funktionsweise erklären. Da der `OfflinePacker` immer im gesamtem Kistensatz nach einem neuem Platz für eine Kiste sucht, kann er die Anzahl der Kisten ausnutzen. Der aufteilende Packer packt jedoch immer t, also 8 bzw. 10 Kisten ineinander. Diese Kistengruppen haben jeweils etwa die gleiche Dichte, also auch der gesamte Kistensatz.

Laufzeit [in Millisekunden]

Algorithmus	**800 Kisten**	**1.600 Kisten**	**3.200 Kisten**
Kleine Kisten			
AufteilenderPacker ($t = 8$)	346	670	1.492
AufteilenderPacker ($t = 10$)	4.074	8.352	18.198
OfflinePacker	348	1.340	5.290
OfflinePacker (eine Strategie)	373	2.237	7.056
Große Kisten			
AufteilenderPacker ($t = 8$)	413	837	1.719
AufteilenderPacker ($t = 10$)	4.074	8.352	18.198
OfflinePacker	182	633	2.302
OfflinePacker (eine Strategie)	180	669	2.527

Bei dem aufteilenden Packer lässt sich ganz klar die lineare Laufzeitkomplexität beobachten. Der Laufzeitsanstieg von 346 ms auf 670 ms ($346 \cdot 2 = 692$) respektive 1492 ms ($670 \cdot 2 = 1340$) bei jeweils verdoppelter Anzahl Kisten ist hierfür ein guter Beleg.

Außerdem lässt sich bereits - trotz absolut schnellerer Laufzeit - die schlechtere Komplexitätsstufe von `OfflinePacker` erkennen. Denn wie erkennbar, steigt die Laufzeit nicht linear mit der Kistenanzahl, sondern quadratisch. Der Algorithmus braucht 348 ms bei 800 kleinen Kisten, 1340 ($\approx 348 \cdot 4 = 1392$) ms bei 1.600 Kisten sowie 5290 ($\approx 1340 \cdot 4 = 5360$) ms bei 3.200 Kisten. Bei Verdopplung der Kisten braucht der `OfflinePacker` also etwa viermal so lange.

4.6. Riesige Anzahl Kisten

Abschließend vergleichen wir noch kurz das Laufzeitverhalten bei 10.000 respektive 80.000 Kisten. Ich entschied mich für den Teilungsgrad $t = 9$, da mit $t = 10$ kaum bessere Ergebnisse erzielt werden können, aber eine deutlich höhere Laufzeit erfordert wird. Wir testen jeweils nur noch mit zwei Wiederholungen.

Komprimierungsgrad

Algorithmus	**10.000 Kisten**	**80.000 Kisten**
Kleine Kisten		
AufteilenderPacker ($t = 9$)	1,43	1,43
OfflinePacker	4,02	9,07
Große Kisten		
AufteilenderPacker ($t = 9$)	1,44	1,44
OfflinePacker	9,17	16,89

Bei dem `OfflinePacker` lassen sich sogar Komprimierungsgrade bis deutlich über 10 beobachten. Das heißt, bei 80.000 Kisten, deren Größen zufällig von 1 x 1 x 1 bis 1000 x 1000 x 1000 verteilt sind, lässt sich der Platzverbrauch auf weniger als ein Zehntel reduzieren. Auch bei kleineren Kisten, ist bei über 10.000 Kisten mit etwa einem Viertel des Anfangsvolumen als Endvolumen zu rechnen.

Der aufteilende `Kistenpacker` erreicht wie erwartet weiterhin einen Komprimierungsgrad von ca. 1,4. Dieser Komprimierungsgrad lässt sich in der Regel mit jeder etwas größeren Kistenanzahl erreichen.

Laufzeit [in Millisekunden]

Algorithmus	**10.000 Kisten**	**80.000 Kisten**
Kleine Kisten		
AufteilenderPacker ($t = 9$)	14.620	113.671
OfflinePacker	56.835	2.024.376
Große Kisten		
AufteilenderPacker ($t = 9$)	17.142	146.574
OfflinePacker	18.898	934.460

Die Laufzeit des `OfflinePacker` ist nur noch gerade so beherrschbar. 2.024.376 Millisekunden sind eine knappe Stunde. Hier zeigt sich ganz klar die bessere Laufzeitkomplexität des aufteilenden Packers. Die Linearität der Laufzeit lässt noch deutlich höhere Kistenanzahlen zu, wie wir später nochmal sehen werden.

4.7. Fazit

Zusammenfassend lässt sich sagen, dass der `OptimalPacker` für Kistenanzahlen bis 12 der beste ist, denn er liefert garantiert optimale Ergebnisse. Ab der 13. Kiste ist seine Laufzeit jedoch nicht mehr kontrollierbar. Für Kistenzahlen von 13 bis 3.200 empfiehlt sich, den `OfflinePacker` mit Standard-Strategien zu benutzen. Er bietet hier gute Komprimierungsgrade und trotzdem akzeptable Laufzeit. Der Algorithmus `AufteilendePacker` lässt sich nur sinnvoll ab 3.200 Kisten einsetzen, da er deutlich schlechtere Komprimierungsgrade als der `OfflinePacker` liefert. Er hat jedoch den Vorteil der linearen Laufzeitkomplexität und somit kann er auch größere Kistenanzahlen in praktikabler Zeit bewältigen. Um den Komprimierungsgrad zu erhöhen, müsste jedoch der Teilungsfaktor erhöht werden, was wiederum die Laufzeit dramatisch wachsen lassen würde. Bemerken muss man hier jedoch auch, dass sich der aufteilende Packer auch über mehrere Prozessoren skalieren lässt. Dies ermöglicht Berechnungen mit bis zu über hunderttausend Kisten.

Zuletzt soll noch einmal demonstriert werden, was die lineare Komplexität in der Praxis heißt. Nachfolgend ein Benchmark des Algorithmus' `AufteilenderPacker` mit $t = 8$ und 888.888 Kisten.

```
scala> KUtils.bench(AufteilenderPacker(8), kisten(888888), 1, false)
Packer: AufteilenderPacker(8)
Wiederholungen  Komprimierungsgrad      Zeit
1               1,4                     1716431
res5: (BigDecimal, Long) = (1.401,1716431)
```

In Scala liefert jeder Ausdruck einen Wert zurück. Da in dem obigem Aufruf der Wert jedoch keiner Variable explizit zugewiesen wird, wird der Wert einer generierten Variable `resX` zugewiesen, wobei `X` mit jedem solchen Aufruf inkrementiert wird.

Wie zu sehen, wurden 1.716.431 Millisekunden, also etwa 1.716 Sekunden benutzt. Dies ist eine knappe halbe Stunde und somit noch völlig im machbaren Bereich.

5. Programmnutzung

Die Nutzung des Programms erfolgt primär über eine Scala-Console mit richtig eingestelltem Classpath. Um dies einfach zu erreichen, empfehle ich, die Scala-Console nach Anleitung im Kapitel “Allgemeines” (A. 5) zu starten. Dann haben Sie auch bereits alle nötigen Module, Klassen, etc. importiert.

5.1. Erzeugen der Kisten

Kisten können manuell eingegeben werden, aber auch zufällig erzeugt werden. Zur manuelle Eingabe gibt man die 3 Maße in Klammern mit `x` getrennt an. Dafür geben Sie beispielsweise folgendes ein.

```
scala>val kiste = (8 x 8 x 4)
kiste: de.voodle.tim.bwinf.kisten.KisteLeer = KisteLeer(8,8,4)
```

Um zufällige Kisten zu erzeugen, bedient man sich der Methode `zufallKisten` aus `StandardKisten`. Hierbei müssen die Maximalwerte der Maße angegeben werden. Die Methode erzeugt eine unendliche Liste, also muss noch angeben werden, wie viele Kisten erzeugt werden sollen. Danach sollte dies mit `toList` in eine normale Scala-Liste umgewandelt werden. Um beispielsweise 800 Kisten mit Maximalwerten von 80 in jeder Dimension zu erzeugen, geht man wie folgt vor.

```
scala> val kisten = zufallKisten(80,80,80) take 800 toList
kisten: List[de.voodle.tim.bwinf.kisten.KisteLeer] =
  List(KisteLeer(59,26,6),  KisteLeer(79,33,32), KisteLeer(78,34,25),
       KisteLeer(79,13,1),  KisteLeer(67,21,7),  KisteLeer(54,2,1),
       KisteLeer(49,41,13), KisteLeer(54,23,11), KisteLeer(57,53,12),
       KisteLeer(61,37,37), KisteLeer(70,64,4),  KisteLeer(77,28,3),
       KisteLeer(73,67,34), KisteLeer(71,55,28), KisteLeer(44,25,9),
       KisteLeer(52,28,19), KisteLeer(31,22,10), KisteLeer(54,23,16),
       KisteLeer(50,33,15), KisteLeer(69,15,6),  KisteLeer(72,71,53),
       KisteLeer(53,50,33), KisteLeer(78,67,32), KisteLeer(74,21,1),
       KisteLeer(60,10,3),  KisteLeer(59,28,13), KisteLeer(58,44,12),
       KisteLeer(48,41,5),  KisteLeer(75,70,63), KisteLeer(59,46,43),
       KisteLeer(51,32,31), KisteLeer(44,37,29), KisteLeer(77,57,30),
       KisteLeer(54,50,32), KisteLeer(59,45,21), KisteLeer(40,21,...
```

Es wurden außerdem noch einige gängigen Kistenmaße aus dem Internet[6] recherchiert. Diese finden Sie ebenfalls unter `StandardKisten`. Nachfolgend zeige ich einige Beispiele.

```
scala> val kisten =
         dinodrei :: buecherkarton :: umzugskarton :: paroli :: Nil
kisten: List[de.voodle.tim.bwinf.kisten.KisteLeer] =
  List(KisteLeer(43,35,10), KisteLeer(41,34,32), KisteLeer(63,33,31),
       KisteLeer(61,35,34))
```

[6] http://kartonfritze.de sowie http://umzugskarton.de

5.2. Erstellung der Schachtelung

Die Schachtelung erstellt man, indem die Methode `min` eines `Kistenpacker`s aufgerufen wird. Um die Schachtelung mit `OptimalPacker` auszuführen, gibt man Folgendes in die Konsole ein.

```
scala> val kisten =
         zufallKisten(80,80,80) take 8 toList // Kisten erzeugen
kisten: List[de.voodle.tim.bwinf.kisten.KisteLeer] =
  List(KisteLeer(72,69,21), KisteLeer(67,38,28), KisteLeer(66,57,43),
       KisteLeer(64,42,40), KisteLeer(58,46,25), KisteLeer(70,53,37),
       KisteLeer(28,15,1),  KisteLeer(45,12,8))

scala> val min = OptimalPacker min kisten
min: de.voodle.tim.bwinf.kisten.Kistensatz =
{
KisteHalb(58,46,25)
 \-KisteLeer(45,12,8),
KisteVoll(66,57,43)
 \-KisteLeer(64,42,40)
 \-KisteLeer(28,15,1),
KisteHalb(70,53,37)
 \-KisteLeer(67,38,28),
KisteLeer(72,69,21)
}
```

Um mit dem aufteilendem `Kistenpacker` Kistenschachtelungen zu erstellen, muss noch der Teilungsfaktor angegeben werden.

```
scala> val min = AufteilenderPacker(5) min kisten // Teilungsfaktor t=5
min: de.voodle.tim.bwinf.kisten.Kistensatz =
{
KisteHalb(64,42,40)
 \-KisteLeer(45,12,8),
KisteVoll(66,57,43)
 \-KisteLeer(58,46,25)
 \-KisteLeer(28,15,1),
KisteHalb(70,53,37)
 \-KisteLeer(67,38,28),
KisteLeer(72,69,21)
}
```

Für den `OfflinePacker` können Strategien angegeben werden. Bei Eingabe einer leeren Parameterliste werden einfach die Standard-Strategien benutzt. Außerdem kann angegeben werden, ob sortiert werden soll oder nicht. In dem folgendem Beispiel werden alle von mir implementierten Strategien benutzt, sowie wird die Sortierung nicht benutzt. Es kann jedoch auch die Reihenfolge variiert werden.

```
scala> val strategien = FindeHalbleeren   :: FindeGroesserenLeeren ::
                        FindeZwischenraum :: FindeKleinereWurzel :: Nil
strategien: List[de.voodle.tim.bwinf.kisten.Strategie] =
  List(FindeHalbleeren, FindeGroesserenLeeren,
       FindeZwischenraum, FindeKleinereWurzel)

scala> val min =OfflinePacker(strategien, sortieren = false) min kisten
min: de.voodle.tim.bwinf.kisten.Kistensatz =
{
KisteVoll(66,57,43)
 \-KisteLeer(64,42,40)
 \-KisteLeer(28,15,1),
KisteLeer(67,38,28),
KisteVoll(70,53,37)
 \-KisteLeer(58,46,25)
 \-KisteLeer(45,12,8),
KisteLeer(72,69,21)
}
```

5.3. Benchmarking eines Algorithmus'

5.3.1. Volumina

Hier wird beschrieben, wie Volumina von Kistensätzen und Listen von Kisten berechnet werden können. Dies macht insbesondere Sinn, um den Komprimierungsgrad, also Verhältnis von Anfangsvolumen und Endvolumen zu berechnen.

Volumen einer Kistenliste Angenommen, wir haben in `kisten` eine `List` von `Kisten`, dann kann das Volumen wie folgt berechnet werden. Die `kisten` sind hierbei die oben in 5.2 erzeugten.

```
scala> val v = kisten.map(_.v).sum
v: Int = 653612
```

Genauso kann mit einem `Set`, also einer Menge von Kisten verfahren werden.

Volumen eines Kistensatzes Wenn `ks` ein Kistensatz ist, dann lässt sich das Volumen direkt abfragen.

```
scala> val kv = ks.v
kv: Int = 286593
```

Komprimierungsgrad Nun haben wir das Startvolumen v berechnet, sowie das Endvolumen kv. Die Berechnung des Komprimierungsgrad ist schließlich nur noch eine Division. Allerdings empfiehlt sich `Double` zu benutzen, um Fließkommazahlen zu erhalten.

```
scala> val komprimierungsgrad = v.toDouble / kv
komprimierungsgrad: Double = 2,2806279
```

5.3.2. Benchmarking

Um möglichst gute Vergleiche zwischen Algorithmen liefern zu können, wurde zum Benchmarking die Methode `def bench(Kistenpacker, () => Seq[KisteLeer], Int, Boolean): (BigDecimal, Long)` im Modul `KUtils` implementiert. Diese gibt zu einem `Kistenpacker`, einer Funktion zur Erzeugung von Kisten sowie einer Anzahl von Wiederholungen den durchschnittlich erreichten Komprimierungsgrad sowie die benötigte Durchschnittszeit in Millisekunden zurück. Zusätzlich kann die Ausgabe beschränkt werden, indem zusätzlich `false` am Ende der Parameterliste übergeben wird. Beispielsweise kann so ein `OfflinePacker` nur mit der Strategie `FindeHalbleeren` der 80 Kisten mit zufälliger Größe packt, wie folgt getestet werden.

```
scala> val (packungsdichte, zeit) =
  KUtils.bench(OfflinePacker(FindeHalbleeren :: Nil),
               () => zufallKisten(80,80,80) take 80 toList, 2)
Packer: OfflinePacker(OnlineAlgo({

},List(FindeHalbleeren)),true)
Minimum: {
KisteLeer(18,15,5),
[..] // Kisten ausgelassen
KisteLeer(79,50,38)
}
Minimum: {
KisteLeer(18,12,6),
[..] // Kisten ausgelassen
KisteLeer(79,72,38)
}
```

```
17 Wiederholungen  Komprimierungsgrad      Zeit
18 2               1                       5
19 packungsdichte: BigDecimal = 1
20 zeit: Long = 5
```

Nur die Strategie `FindeHalbleeren` zu benutzen, ist natürlich nicht sinnvoll, denn ohne zusätzliche Strategie wird nie eine halbleere Kiste erzeugt. Der Komprimierungsgrad ist und bleibt einfach 1. Hier dient das Beispiel lediglich zur Demonstration der Benutzung.

6. Programmtext

Alle Quelldateien dieser Aufgabe finden sich auf der CD unter `Aufgabe1/src/`.

6.1. Kisten

```
1  package de.voodle.tim.bwinf.kisten
2  import math._
3
4  sealed abstract class Kiste extends Ordered[Kiste] {
5    val a,b,c: Int
6    final val v = a*b*c // cache it!
7
8    final def +<(der: Kiste) =
9      if(this >~ der)   // Passt der hinein?-
10       this +<< der  // Dann packe ihn ein!
11     else Set.empty // Sonst lass' es sein.
12
13   protected def +<<(der:Kiste): Set[Kiste]
14
15   // Folgende Methoden prüfen unabhänig vom Inhalt!
16   final def >~(der: Kiste) = a > der.a && b > der.b && c > der.c
17   final def >=~(der: Kiste) = a >= der.a && b >= der.b && c >= der.c
18
19   // Benutzung intern.
20   protected[kisten] def <(x: Int, y: Int, z: Int) =
21     a < x && b < y && c < z
22   protected[kisten] def tuple_/:[T](start: T)(f: (T, Int,Int,Int) => T) =
23     f(f(f(f(f(f(start, a,b,c), a,c,b), b,a,c), b,c,a), c,a,b), c,b,a)
24
25   final def compare(der: Kiste) =
26     if(a != der.a) a - der.a
27     else if(b != der.b) b - der.b
28     else if(c != der.c) c - der.c
29     else vergleichInhalt(der)
30   protected def vergleichInhalt(der: Kiste): Int
31
32   final override def toString = baumString(0)
33   private def baumString(lvl: Int): String = {
34     val pref = "␣" * lvl + (if(lvl > 0) "\\-" else "")
35     val selbst = getClass.getSimpleName + (a,b,c)
36     pref + selbst + (if(kinder.isEmpty) "" else "\n") +
37       kinder.map(_.baumString(lvl + 1)).
38              mkString("\n")
39   }
40
41   override def equals(that: Any) = that match {
42     case der: Kiste =>
43       getClass() == der.getClass() &&
44         a == der.a && b == der.b && c == der.c
45     case _ => false
46   }
47   override def hashCode = 31*(31*(31*(31*(31 + v) + a) + b) + c)
48
49   def finde(f: Kiste => Boolean): List[Kiste] =
50     if(f(this)) this :: Nil
51     else {
52       val kinderPfade = kinder.map(_.finde(f))
53       kinderPfade.find(!_.isEmpty) map {
54         path => this :: path
55       } getOrElse Nil
56     }
57
```

```
58    def kinder: Seq[Kiste]
59    def istLeer = kinder.isEmpty
60    def alsLeer = KisteLeer(a,b,c)
61    def toStream: Stream[Kiste] =
62      Stream(this).append(kinder flatMap (_.toStream))
63
64    def *(anzahl: Int) = List.fill(anzahl)(this)
65  }
66
67  case class KisteLeer private[kisten](a: Int, b: Int, c: Int)
68    extends Kiste {
69
70    def kinder = Seq.empty
71
72    def +(der: Kiste) = KisteHalb(a,b,c, der)
73    protected def +<<(der: Kiste) = Set(KisteHalb(a,b,c, der))
74
75    protected def vergleichInhalt(der: Kiste) = der match {
76      case _: KisteLeer => 0//Zwei leere Kisten gleicher Größe sind die gleichen
77      case _ => -1               //Sonst muss dieser Leere als Kleinerer weichen
78    }
79
80    override val hashCode = // Nutze den Companion, für die Hashfunktion.
81      31*(31*(31* + super.hashCode) + KisteLeer.hashCode)
82
83    // Überschreibe wegen Types!
84    override def alsLeer = this
85    override def *(anzahl: Int) = List.fill(anzahl)(this)
86  }
87
88  case class KisteHalb
89    private[kisten](a: Int, b: Int, c: Int, links: Kiste) extends Kiste {
90
91    override def istLeer = false
92    def kinder = Seq(links)
93    def freiFür(der: Kiste) = (false tuple_/: links) {
94        (prev, x,y,z) => prev ||
95          der < (a-x,b,c) ||
96          der < (a,b-y,c) ||
97          der < (a,b,c-z)
98        }
99
100   def ersetzeLinks(nl: Kiste) = KisteHalb(a,b,c, nl)
101
102   def +(der: Kiste): KisteVoll = Kiste(a,b,c, links, der)
103
104   protected def +<<(der: Kiste) = {
105       val neueLinks = links +< der
106       val kisten: Set[Kiste] = neueLinks.map(ersetzeLinks(_))
107       if(this freiFür der)
108         kisten + (this + der)
109       else
110         kisten
111   }
112   protected def vergleichInhalt(der: Kiste) = der match {
113       case leer: KisteLeer => 1       // Ein leerer! Ach wie Tolle!
114       case halb: KisteHalb =>
115         this.links compare halb.links // !Hier! spielt der Inhalt eine Rolle!
116       case _ => -1                    // Schau! Da bleibt nur noch der Volle..
```

```
117     }
118
119   override val hashCode = // wand sollte nie <0 sein
120     31*(31*(31*(31* + super.hashCode) + links.hashCode) + KisteHalb.hashCode)
121 }
122
123 case class KisteVoll
124   private[kisten](a: Int, b: Int, c: Int, links: Kiste, rechts: Kiste) extends Kiste {
125
126   override def istLeer = false
127   def kinder = Seq(links, rechts)
128
129   def ersetzeLinks(nl: Kiste)  = Kiste(a,b,c, nl, rechts)
130   def ersetzeRechts(nr: Kiste) = Kiste(a,b,c, links,  nr)
131
132   private val linksGleichRechts = links == rechts
133   protected def +<<(der: Kiste) = {
134     val linkeSeite: Set[Kiste] = (links +< der).map(ersetzeLinks(_))
135     if(linksGleichRechts)
136       linkeSeite
137     else {
138       val rechteSeite = (rechts +< der).map(ersetzeRechts(_))
139       linkeSeite ++ rechteSeite
140     }
141   }
142   protected def vergleichInhalt(der: Kiste) = der match {
143     case voll: KisteVoll =>                              // Der Inhalt entscheide!
144       val linksDiff = this.links compare voll.links // Prüf erst die linke Seite!
145       if(linksDiff != 0) linksDiff              // Sind es auch nicht die gleichen?
146       else this.rechts compare voll.rechts      // Dann müssen die Rechten reichen
147     case _ => 1                             // Sonst muss der kleine and're weichen!
148   }
149
150   override val hashCode =
151     31*(31*(31*(31*(31 + super.hashCode) + links.hashCode) + rechts.hashCode)
152         + KisteVoll.hashCode)
153 }
154
155 object Kiste {
156   def ordne(x: Int, y: Int, z: Int) = {
157     var (a,b,c) = (x,y,z)
158     var tmp = 0
159     if(a < b) { tmp = a; a = b; b = tmp }
160     if(a < c) { tmp = a; a = c; c = tmp }
161     // es gilt jetzt: a >= b && a >= c
162     if(b < c) { tmp = b; b = c; c = tmp }
163     // es gilt jetzt: a >= b >= c
164     // also wenn eine Zahl nicht positiv ist, dann auf jeden Fall auch c
165     if(c < 0)
166      throw new IllegalArgumentException("Negative␣Werte␣sind␣nicht␣erlaubt!")
167     else
168       (a,b,c)
169   }
170   // Objekterzeuger [Hilfs-]Methoden:
171   def apply(x: Int, y: Int, z: Int) = {
172     val (a,b,c) = ordne(x,y,z)
173     new KisteLeer(a,b,c)
174   }
175   def apply(x: Int, y: Int, z: Int, links: Kiste) = {
```

```
176     val (a,b,c) = ordne(x,y,z)
177     new KisteHalb(a,b,c, links)
178   }
179
180   def apply(x: Int, y: Int, z: Int, links: Kiste, rechts: Kiste) = {
181     val (a,b,c) = ordne(x,y,z)
182     // links muss >= rechts sein!
183     if(links >= rechts) new KisteVoll(a,b,c, links, rechts)
184     else                new KisteVoll(a,b,c, rechts, links)
185   }
186
187   object Ordnung {
188     object nachVolumen extends Ordering[Kiste] {
189       def compare(dieser: Kiste, anderer: Kiste) = dieser.v - anderer.v
190     }
191     object eindeutig extends Ordering[Kiste] {
192       def compare(dieser: Kiste, anderer: Kiste) = dieser compare anderer
193     }
194   }
195
196   implicit def intToPartialKiste(a: Int): PartialKisteA = PartialKisteA(a)
197 }
198
199 // Helper für schöne Syntax:
200 case class PartialKisteA(a: Int) {
201   def x(b: Int) = PartialKisteAB(a,b)
202 }
203 case class PartialKisteAB(a: Int, b: Int) {
204   def x(c: Int) = Kiste(a,b,c)
205 }
```

6.2. Kistensatz

```
1  package de.voodle.tim.bwinf.kisten
2
3  import scala.collection.immutable.SortedSet
4  import scala.collection.immutable.TreeMap
5
6  case class Kistensatz (kistenBaum: TreeMap[Kiste, Int], v: Int, length: Int)
7    extends Ordered[Kistensatz] {
8    import Kistensatz._
9
10   def find(f: Kiste => Boolean): List[Kiste] =
11     kistenBaum.find(k => f(k._1)) map (_._1 :: Nil) getOrElse // Gibt es Wurzel?
12     (((None: Option[List[Kiste]]) /: kistenSet) { // Sonst suche in Elementen.
13       (vorher, kiste) =>
14         if(vorher.isEmpty) {
15           val pfad = kiste.finde(f)
16           if(pfad.isEmpty)
17             None
18           else
19             Some(pfad)
20         } else vorher
21     } getOrElse Nil)
22
23   def -(der: Kiste) = Kistensatz(subFromTree(kistenBaum, der), v-der.v, length-1)
24   def +(der: Kiste) = Kistensatz(  addToTree(kistenBaum, der), v+der.v, length+1)
25
26   def +<(der: Kiste): Set[Kistensatz] =
27     (Set[Kistensatz]() /: kistenBaum) {
28         case (saetze, (k, _)) => {
29           val treeOhneK = subFromTree(kistenBaum, k)
30           val gepackt = (k +< der)
31           val ms = gepackt.map(g =>
32             Kistensatz(addToTree(treeOhneK, g), v, length)) // k.v == g.v
33           saetze ++ ms
34         }
35       }
36   def ++<(der: Kiste): Set[Kistensatz] =
37     (this +< der) + (this + der)
38
39   def neben(der: Kistensatz) =
40     new AneinandergereihterKistensatz(der :: this :: Nil)
41
42   def compare(der: Kistensatz) = Kistensatz.Ordnung.eindeutig.compare(this, der)
43
44   override def equals(other: Any) =
45     other.isInstanceOf[Kistensatz] && equals(other.asInstanceOf[Kistensatz])
46   def equals(der: Kistensatz) = // Schneller HashCode basierter Check
47     hashCode == der.hashCode && kistenBaum == der.kistenBaum
48
49   override val hashCode = 41* (43 /: kistenBaum) { 47* _ + _.hashCode } + v
50
51   def kistenSet: SortedSet[Kiste] = SortedSet(kistenBaum.map(_._1).toSeq: _*)
52   def kisten = kistenBaum.flatMap((ki) => ki._1 * ki._2).toSeq.sorted
53   def toStream = kisten.flatMap(_.toStream).toStream
54   override def toString = kisten.mkString("{\n", ",\n", "\n}")
55 }
56
57 object Kistensatz {
```

```
58    def addToTree(baum: TreeMap[Kiste, Int], karton: Kiste) =
59      baum.updated(karton, baum.getOrElse(karton, 0) + 1)
60
61    def subFromTree(baum: TreeMap[Kiste, Int], karton: Kiste) =
62      baum.get(karton) match {
63        case Some(i) => if(i > 1) baum.updated(karton, i-1) else (baum - karton)
64        case _ => baum
65      }
66    private def createTree(tree: TreeMap[Kiste, Int], liste: Seq[Kiste]): TreeMap[Kiste, Int] =
67      if(liste.isEmpty) tree
68      else createTree(addToTree(tree, liste.head), liste.tail)
69
70    def apply(kisten: Seq[Kiste]): Kistensatz =
71        apply(createTree(TreeMap.empty, kisten))
72    def apply(kistenBaum: TreeMap[Kiste, Int]): Kistensatz =
73      Kistensatz(kistenBaum,
74                 (0 /: kistenBaum) { (v,ki) => v + ki._1.v * ki._2 },
75                 (0 /: kistenBaum) { (n,ki) => n + ki._2 } )
76
77    object Ordnung {
78      val nachVolumen = new Ordering[Kistensatz]() {
79        def compare(x: Kistensatz, y: Kistensatz) = x.v compare y.v
80      }
81      val eindeutig = new Ordering[Kistensatz]() {
82        def compare(x: Kistensatz, y: Kistensatz) =
83          if(x.v != y.v) x.v - y.v
84          else compare(x.kisten, y.kisten)
85
86        private def compare(x: Iterable[Kiste], y: Iterable[Kiste]): Int =
87          if(x.isEmpty && y.isEmpty) 0
88          else if(x.isEmpty) -1
89          else if(y.isEmpty) 1
90          else {
91            val diff = x.head compare y.head
92            if(diff == 0) compare(x.tail, y.tail)
93            else diff
94          }
95      }
96    }
97  }
```

6.3. Kistenpacker

```scala
package de.voodle.tim.bwinf.kisten

trait Kistenpacker {
  def min(kisten: Seq[KisteLeer]): Kistensatz
  // Sortiert wird von Groß nach Klein!
  // D'rum muss die Ordnung 'falsch'rum sein
  def sortiere(input: Seq[KisteLeer]): List[KisteLeer] =
    input.toList.sorted(Kiste.Ordnung.nachVolumen.reverse)
}
```

6.3.1. OptimalPacker

```scala
package de.voodle.tim.bwinf.kisten

case object OptimalPacker extends Kistenpacker {
  def min(kistenListe: Seq[KisteLeer]) =
    packe(kistenListe) min Kistensatz.Ordnung.nachVolumen

  def packe(kistenListe: Seq[KisteLeer]) =
    (Set[Kistensatz]() /: sortiere(kistenListe)) ( packSchritt )

  protected def packSchritt(sätze: Set[Kistensatz], kiste: KisteLeer) =
    if(sätze.isEmpty)
      Set(Kistensatz(kiste :: Nil)) // KistenSatz nur mit der Kiste
    else for { satz <- sätze
               neuerSatz <- satz ++< kiste } yield neuerSatz
}
```

6.3.2. AufteilenderPacker

```scala
package de.voodle.tim.bwinf.kisten

case class AufteilenderPacker (t: Int = 15) extends Kistenpacker {
  def min(kisten: Seq[KisteLeer]): Kistensatz = {
    val kistengruppen = teileKisten(kisten)
    val kistensätze = for {
      kistengruppe <- kistengruppen
    } yield OptimalPacker min kistengruppe

    kistensätze reduceLeft { _ neben _ }
  }

  private def teileKisten(kisten: Seq[KisteLeer]) = {
    val m = (kisten.length.toDouble / t).ceil.toInt
    val startListen = Vector.fill(m)(List.empty[KisteLeer])
    ((startListen, 0) /: kisten) {
      case ((listen, idx), kiste) =>
        val liste = listen(idx)
        (listen updated (idx, kiste :: liste), (idx+1)%m)
    } match {
      case (listen, _) => listen
    }
  }
}
```

6.3.3. OfflinePacker und OnlineAlgo

```
1  package de.voodle.tim.bwinf.kisten
2
3  /** Gliedert den OnlineAlgo in die Packer Hierarchie */
4  case class OfflinePacker(onlinePacker: OnlineAlgo, sortieren: Boolean)
5    extends Kistenpacker {
6    def min(kistenSeq: Seq[KisteLeer]) = {
7      val kisten = if(sortieren) sortiere(kistenSeq) else kistenSeq
8      (onlinePacker ++ kisten).kisten
9    }
10 }
11 object OfflinePacker {
12   def apply(strategien: List[Strategie], sortieren: Boolean = true): OfflinePacker =
13     OfflinePacker(OnlineAlgo(strategien), sortieren)
14   def apply(sortieren: Boolean): OfflinePacker =
15     OfflinePacker(OnlineAlgo.standardStrategien, sortieren)
16   def apply(): OfflinePacker =
17     OfflinePacker(OnlineAlgo.standardStrategien)
18 }
19 case class OnlineAlgo(kisten: Kistensatz, strategien: List[Strategie]) {
20   /** Einfach alle nacheinander hineinfügen! */
21   def ++ (die: Seq[KisteLeer]) = (this /: die)(_ + _)
22
23   def + (der: KisteLeer): OnlineAlgo = {
24     val neueKisten = (Option.empty[Kistensatz] /: strategien) {
25         // wenn es  vorher gibt, dann vorher, sonst diese strategie benutzen!
26          (vorher, strat) => vorher orElse strat(kisten)(der)
27       } getOrElse (kisten + der) // sonst aneinanderreihen, wie einfach :D
28     OnlineAlgo(neueKisten, strategien)
29   }
30 }
31 object OnlineAlgo { // Hilfsmethoden zum schnellen erzeugen! :)
32   def apply(): OnlineAlgo = apply(Kistensatz(Nil))
33   def apply(strategien: List[Strategie]): OnlineAlgo =
34     OnlineAlgo(Kistensatz(Nil), strategien)
35   def apply(kistenSatz: Kistensatz): OnlineAlgo =
36     OnlineAlgo(kistenSatz, standardStrategien)
37
38   def standardStrategien =
39     FindeGroesserenLeeren :: // (2)
40     FindeZwischenraum     :: // (3)
41     FindeHalbleeren       :: // (1)
42     FindeKleinereWurzel :: // (4)
43     Nil
44 }
```

6.3.4. Strategien

```
package de.voodle.tim.bwinf.kisten

abstract trait Strategie {
  // Syntactic sugar :)
  def apply(kisten: Kistensatz)(der: KisteLeer) =
    finde(kisten)(der)

  protected def finde(kisten: Kistensatz)(der: KisteLeer): Option[Kistensatz]
}

object Strategie {
  private[kisten] def pfadErsetzer(pfad: List[Kiste]): (Kiste,Kiste) => Kiste = {
      case (kg: KisteHalb, kl) => kg ersetzeLinks kl
      case (kg: KisteVoll, kl) =>
        if(pfad.contains(kg.links)) kg ersetzeLinks  kl
        else                        kg ersetzeRechts kl
      case _ =>
        throw new IllegalArgumentException("Leere Kisten im Pfad nicht erlaubt!")
    }
}
import Strategie._

case object FindeHalbleeren extends Strategie {
  protected def finde(kisten: Kistensatz)(der: KisteLeer) =
    kisten.find {
      case kh: KisteHalb =>
        (kh >~ der) && kh.freiFür(der)
      case _ => false
    } match {
      case Nil => None
      case pfad =>
        val alteKiste = pfad.head
        val neueKiste = (pfad :\ (der: Kiste)) { (_, _) match {
            case (kg: KisteHalb, kl: KisteLeer) => kg + kl
            case (k1, k2) => pfadErsetzer(pfad)(k1, k2)
          }}
        Some(kisten - alteKiste + neueKiste)
    }
}
case object FindeGroesserenLeeren extends Strategie {
  protected def finde(kisten: Kistensatz)(der: KisteLeer) =
    kisten.find {
      big => (big >~ der) && big.istLeer
    } match {
      case Nil => None
      case pfad =>
        val alteKiste = pfad.head
        val neueKiste = (pfad :\ (der: Kiste)) { (_, _) match {
            case (kg: KisteLeer, kl: KisteLeer) => kg + kl
            case (k1, k2) => pfadErsetzer(pfad)(k1, k2)
          }}
        Some(kisten - alteKiste + neueKiste)
    }
}
case object FindeZwischenraum extends Strategie {
  protected def finde(kisten: Kistensatz)(der: KisteLeer) =
    kisten.find { k => (k >~ der) && (k match {
```

```
58        case kh: KisteHalb =>//Es ist sicher noch Platz für einen Zwischenraum
59            (der >~ kh.links)//Aber kh.links muss auch in der passen!
60        case kv: KisteVoll =>//Nur dann ist Platz, wenn 'der' noch neben den anderen reinpasst.
61          val links  = Kiste(kv.a,kv.b,kv.c, kv.links)
62          val rechts = Kiste(kv.a,kv.b,kv.c, kv.rechts)
63          ((der >~ kv.links)  && rechts.freiFür(der)) ||
64          ((der >~ kv.rechts) && links.freiFür(der))
65        case _ => false
66       })
67      } match {
68        case Nil => None
69        case pfad =>
70          val alteKiste = pfad.head
71          val neueKiste = (pfad :\ (der: Kiste)) {
72            case (kg: KisteHalb, kl: KisteLeer) =>
73              kg ersetzeLinks (kl + kg.links)
74            case (kg: KisteVoll, kl: KisteLeer) =>
75              if(kl >~ kg.links) kg ersetzeLinks  (kl + kg.links)
76              else               kg ersetzeRechts (kl + kg.rechts)
77            case (k1, k2) => pfadErsetzer(pfad)(k1, k2)
78          }
79          Some(kisten - alteKiste + neueKiste)
80      }
81  }
82  case object FindeKleinereWurzel extends Strategie {
83    protected def finde(kisten: Kistensatz)(der: KisteLeer) =
84      kisten.kistenSet.find(der >~ _) map {
85        kWurzel => kisten - kWurzel + (der + kWurzel)
86      }
87  }
```

6.4. Helfer

6.4.1. KUtils

```
package de.voodle.tim.bwinf.kisten

import java.math.RoundingMode.{UP => ROUND_UP}, java.math.MathContext
import java.text.DecimalFormat

object KUtils {
  import StandardKisten._ // Standardkartongrößen
  import KistenLogger._, LogStufe._

  private val logger = KistenLogger(print, getNextFileWriter)
  import logger.logge

  private implicit val mathContext = new MathContext(4, ROUND_UP)

  def bench(packer: Kistenpacker, kistenFunktion: ()=>Seq[KisteLeer],
            wiederholungen: Int, verbose: Boolean = true): (BigDecimal, Long) = {
    logge("Packer:␣" + packer)
    var ergebnisse: List[((BigInt, BigInt), Long)] = Nil
    var i = 0
    while(i < wiederholungen) {
      val kisten = kistenFunktion()

      val (min, zeit) = bench(packer min kisten)
      logge("Minimum:␣" + min)(verbose)

      val v0 = kisten.map(i => BigInt(i.v)).sum
      val v1 = min.kisten.map(i => BigInt(i.v)) reduceLeft (_ + _)
      ergebnisse ::= ((v0,v1), zeit)
      i += 1
    }
    val gesamtVolumen = ergebnisse.map(_._1._1).sum
    val endVolumen    = ergebnisse.map(_._1._2) reduceLeft (_ + _)
    val komprimierung = BigDecimal(
      BigDecimal(gesamtVolumen).bigDecimal
        .divide(BigDecimal(endVolumen).bigDecimal, mathContext)
      )
    val zeiten = ergebnisse map (_._2)
    val zeit = zeiten.sum / zeiten.length
    val format = new DecimalFormat("##.##")
    logge("Wiederholungen\tKomprimierungsgrad\tZeit")
    logge(wiederholungen + "\t\t" + format.format(komprimierung) +
                           "\t\t\t" + format.format(zeit))
    (komprimierung, zeit)
  }

  private def zufallsKisten(n: Int) = List.fill(n)(zufall)

  // Hilfsmethode zum benchen.
  private def bench[T](packer: =>T): (T, Long) = {
    val startZeit = System.currentTimeMillis
    val kistenSaetze = packer
    val endZeit = System.currentTimeMillis
    (kistenSaetze, endZeit - startZeit)
  }
}
```

6.4.2. StandardKisten

```
package de.voodle.tim.bwinf.kisten

import Kiste._ // Implicits importieren für schöne Syntax

object StandardKisten { // Alle Angaben in cm
  // Quelle: http://umzugskarton.de
  val umzugskarton  = 63 x 31 x  33
  val buecherkarton = 41 x 32 x  34
  val kleiderbox    = 50 x 60 x 135

  // Quelle: http://www.kartonfritze.de
  val pizzakarton = 24 x 24 x 4// .. auch wenn Frau Y. nicht Informatikfreak[in] ist..
  val arthur    = 46 x 38 x 29
  val dinodrei  = 43 x 35 x 10
  val fluschi   = 43 x 34 x 23
  val christoph = 42 x 33 x 37

  // "Ideale Kartongröße für Vertreter des weiblichen Geschlechts." laut Kartonfritze
  val paroli      =  61 x 34 x 35
  val postmeister = 120 x 60 x 60 // Was machen jetzt Postmeisterinnen?
  val chachacha   = 118 x 20 x  9 // Waffenkarton: "Geeignet für alles was passt!"-ach nee!

  private def zufallGen = scala.util.Random
  def zufall: KisteLeer = zufall(60,60,60)
  def zufall(maxA: Int, maxB: Int, maxC: Int): KisteLeer =  // Größer 0!
    zufallGen.nextInt(maxA-1) + 1 x
    zufallGen.nextInt(maxB-1) + 1 x
    zufallGen.nextInt(maxC-1) + 1
  def zufallKisten(maxA: Int, maxB: Int, maxC: Int): Stream[KisteLeer] =
    Stream.cons(zufall(maxA,maxB,maxC), zufallKisten(maxA, maxB, maxC))
}
```

6.4.3. KistenLogger

```
package de.voodle.tim.bwinf.kisten

object LogStufe extends Enumeration {
  type LogStufe = Value
  val Extra    = Value(20)
  val Ergebnis = Value(10)
}
object KistenLogger {
  import java.io.{File, FileWriter}
  val suffix = "_kisten.log"
  def getNextFileWriter =
    Stream from 0 map (i => new File("./" + i + suffix)) find {
      ! _.exists
    } map {
      file => new FileWriter(file)
    } getOrElse (throw new AssertionError("No file found!"))
  implicit def ausgabeAusFileWriter(fw: FileWriter): String => Any = {
    str => fw.write(str); fw.flush
  }
}
case class KistenLogger(printer: String => Any, dateiAusgabe: String => Any) {
  import LogStufe._
  def logge(zeile: =>String)(implicit print: Boolean = true) =
```

```
24      if(print) {
25        val ausgabenZeile = zeile + "\n" // By-name!
26        printer(ausgabenZeile)
27        dateiAusgabe(ausgabenZeile)
28      } else ()
29  }
```

C. Zweite bearbeitete Aufgabe: (2) Containerklamüsel

Inhalt der Aufgabendokumentation

1. Lösungsidee . . . 47
2. Algorithmen und Datenstrukturen . . . 48
2.1. Anmerkung zur Notation . . . 48
2.2. Datenstrukturen . . . 48
2.2.1. Permutation . . . 48
2.2.2. Gleis . . . 48
2.2.3. Kraninstruktionen . . . 49
2.3. Ergebnisoptimaler Algorithmus . . . 49
2.3.1. Entwurf . . . 49
2.3.2. Optimale Ergebnisse . . . 53
2.3.3. Laufzeitverhalten . . . 56
2.4. Ergebnis- und laufzeitoptimaler Algorithmus . . . 58
2.4.1. Verbesserung . . . 58
2.4.2. Optimale Ergebnisse . . . 59
2.4.3. Optimale Laufzeitkomplexität . . . 59
2.4.4. Mögliche Parallelisierung . . . 59
3. Implementierung . . . 60
3.1. Cycler - Berechnung der Zyklen . . . 60
3.2. Instructor - Berechnung der Kraninstruktionen . . . 60
3.3. Gleis - Speichern des Zustands . . . 60
3.4. Instructions - Die Anweisungen . . . 61
3.5. Maschine - Interpretieren der Kraninstruktionen . . . 61
3.6. Utils - Helfende Methoden . . . 61
3.7. ListBuffer - Erweiterung der Standardklasse ListBuffer . . . 61
4. Programmabläufe . . . 62
4.1. Beispiel aus der Aufgabenstellung . . . 63
4.2. Spezialfälle . . . 64
4.3. Zufällig erzeugte Permutationen . . . 69
4.4. Demonstration der Skalierbarkeit . . . 72
5. Programmnutzung . . . 73
5.1. Permutationen erzeugen . . . 73
5.2. Kraninstruktionen erzeugen . . . 73
5.3. Simulation der Kranmaschine . . . 74
5.4. Zeitmessung . . . 74
6. Programmtext . . . 75
6.1. Cycler . . . 76
6.2. Instructor . . . 77
6.3. Gleis . . . 78
6.4. Instructions . . . 79
6.5. Maschine . . . 80
6.6. Utils . . . 82
6.7. ListBuffer . . . 84

1. Lösungsidee

Zunächst lässt sich feststellen, dass die Anordnung der Waggons zu den Containern eine bijektive Abbildung von $[1, n]$ nach $[1, n]$, sprich, eine Permutation der Menge $[1, n]$ ist. Für die Lösung der Aufgabe werden bestimmte Eigenschaften von Permutationen verwendet. 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 ist und was er für die Aufgabe bedeutet, lässt folgende Darstellung veranschaulichen. Hier bilden die Container $2, 5, 4, 8, 1, 2$ einen Zyklus. Ausgehend von Waggonposition 1 wird der Container 2 an die Position 2 transportiert, anschließend der dortigen Container 5 an die Position 5, usw. bis zum Container 1 an die Position 1. Man beachte, dass die Container so "in einem Stück" getauscht und an die richtige Position gebracht werden können.

```
1 1 2 3 4 5 6 7 8 (Waggonposition)
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[1]. Da von Beutelspacher einige hier nicht eingeführten Begriffe verwendet wurden, habe ich diese ersetzt. Alle eigenständigen Änderungen wurden mit doppelten geschweiften Klammern ('{{' und '}}') markiert, um Verwechslung mit den als mathematische Symbole verwendeten eckigen Klammern zu vermeiden. Auslassungen markiere ich mit '{{...}}'

Anfang Zitat

> {{Sei S_n die Menge aller Permutation von $[1, n]$ in sich.}}
> Eine Permutation π{{$\in S_n$}} 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 $i \in${{S_n]}} und eine natürliche Zahl k gibt, so dass die folgenden drei Bedingungen gelten:
> (1) $\pi^k(i) = i$,
> (2) die Elemente $i, \pi(i), \pi^2(i), \ldots, \pi^{k-1}(i)$ sind paarweise verschieden,
> (3) jedes Element, das verschieden von $i, \pi(i), \pi^2(i), \ldots, \pi^{k-1}(i), \pi^k(i)(= i)$ 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
>
> $$\pi = (i\ \pi(i)\ \pi^2(i)\ \ldots\ \pi^{k-1}(i)). \qquad \{\{\ldots\}\}$$
>
> **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 $\pi \in S_n$ gibt es zyklische Permutationen $\zeta_1, \ldots, \zeta_s \in S_n$, so dass folgende Eigenschaften erfüllt sind:*
> *- $\pi = \zeta_1 \cdot \zeta_2 \cdot \ldots \cdot \zeta_s$*
> *- kein Element aus {{$[1, n]$}}, das {{...}} in ζ_i vorkommt, kommt in ζ_j vor $(i, j = 1, \ldots, n, i \neq j)$.* (Das bedeutet: Wenn ein Element $x \in${{$[1, n]$}} in einem Zyklus ζ_i "vorkommt", so wird x von jedem anderen Zyklus ζ_j $(j \neq i)$ fest gelassen.)

Ende Zitat

[1] Definitionen, Sätze und Erklärung übernommen aus Lineare Algebra, Albrecht Beutelspacher, S.174f

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 Teil der Problemlösung 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, usw., 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. Der Sinn dieser verschachtelten Zyklenunterbrechungen ist es, Leerfahrten - und andere vermeidbare Fahrten - zu vermeiden.

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 die Ausgangsposition gefahren werden. Wir werden etwas später sehen, dass dadurch tatsächlich auch immer ein optimaler Weg (zumindest innerhalb eines Zyklus) gefunden werden kann. 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.

2. Algorithmen und Datenstrukturen

2.1. Anmerkung zur Notation

Zur Erläuterung mathematischer Überlegungen wird der $math-Mode$ von TEXbenutzt. Statt beispielsweise `perm` für eine Permutation oder `cycle` für einen oben beschrieben Zyklus als Bezeichner im "code-Mode" benutze ich π respektive ζ im "math-Mode" für korrespondierende mathematische Überlegungen. Es werden also die jeweils passenderen Bezeichner und Symbole verwendet.

Um den Code kompakt zu halten, werden folgende sogenannte "type aliases" verwendet.

`Cycle`	für `List[Int]`
`Cycles`	für `List[Cycle]`

2.2. Datenstrukturen

2.2.1. Permutation

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 geeignet beachtet werden. Der Definitionsbereich ist also um eins "nach links" verschoben. Zur Darstellung einer Permutation im Code wird `perm: Seq[Int]` benutzt, also eine Sequenz (z. B. `Array` oder `List`) von Zahlen benutzt. Um die Zahl `p` zu finden, auf die `i` durch `perm` abgebildet wird, gilt `p = perm(i-1)` und nicht `p = perm(i)`.

2.2.2. Gleis

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

2.2.3. Kraninstruktionen

Es gibt mehrere Befehle, die ein Kran nach der Aufgabenstellung ausführen kann. Ich verwende die folgenden sieben *Kraninstruktionen*.

PutCon	Ablegen des Containers auf den Containerstellplatz unter dem Kran
PutWag	Ablegen des Containers auf den Waggon unter dem Kran
TakeCon	Aufnehmen des Containers von dem Containerstellplatz unter dem Kran
TakeWag	Aufnehmen des Containers von dem Waggon unter dem Kran
Rotate	Drehen des Krankopfes: Die aufgehobenen Container werden jeweils auf die andere Seite gedreht.
MoveRight(x)	Bewegen des Kranes in Richtung Gleisende um x Positionen
MoveLeft(x)	Bewegen des Kranes in Richtung Gleisanfang um x Positionen

Seinen x, y zwei beliebige Indizes dann können die Befehle `MoveRight` bzw. `MoveLeft` zusammengefasst mit `Move(x -> y)` dargestellt werden. Wenn $x < y$ so stellt dies ein `MoveRight` dar, bei $x >= y$ ein `MoveLeft`. Gelesen werden kann `Move(x -> y)` als "bewege Kran von x nach y".

2.3. Ergebnisoptimaler Algorithmus

In diesem Abschnitt wird zunächst ein Algorithmus entworfen, der optimale Ergebnisse (im Sinne von kürzesten Kranwegen) berechnet (2.3.1). Danach wird die Optimalität bewiesen (2.3.2) und anschließend wird das Laufzeitverhalten dieses Algorithmus betrachtet, welches gut, jedoch nicht bestmöglich ist (2.3.3). Im nächsten Abschnitt 2.4 wird ein Algorithmus vorgestellt, der sowohl optimale Ergebnisse berechnet, als auch optimale Laufzeitkomplexität vorweist.

2.3.1. Entwurf

Finden eines Zyklus' Der Entwurf dieses Algorithmus' ergibt sich aus der Lösungsidee. Zunächst wird die Zerlegung in disjunkte Zyklen berechnet. Hierfür wird folgende Hilfsfunktion `cycle` zur Berechnung *eines* Zyklus' verwendet.
Salopp gesagt, hangelt 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' hinter `idx` als Liste. Die Unterfunktion `step` bricht mit der leeren Liste ab, wenn `start` wieder erreicht wird. Andernfalls reiht `step` den aktuellen Wert `idx` vor die restlichen - rekursiv durch `step` - berechneten Zahlen. Die Funktion `cycle` braucht nur mehr `step` mit `start` aufzurufen.

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

Das Ergebnis von `cycle` ist also ein Zyklus der oben beschrieben Form $(i\pi(i)\pi^2(i)\dots\pi^{k-1}(i))$ dargestellt als `start :: perm(start-1) :: perm(perm(start-1)-1) :: ... :: Nil`. Wichtig ist hierbei zu beachten, dass die Waggonnummer, auf den der Container an der Stelle `idx` gebracht werden muss, durch `perm(idx-1)` dargestellt wird.

Finden aller Zyklen 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 mit `find` der neue Startwert gesucht. Dieser ist die kleinste 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 - was bedeutet, dass bereits alle Zahlen in einem Zyklus vorkommen - so wird die Rekursion abgebrochen. Es wird dann die leere Liste `Nil` zurückgegeben.

```
1  def cyclesOf(perm: Seq[Int], handled: Set[Int]): Cycles =
2    // Suchen des Startwertes, dann das Ergebnis matchen
3    (1 to perm.length) find (i => !handled.contains(i)) match {
4        case Some(start) => // Wenn es ein Startwert start gibt
5          val newCycle = cycle(perm, start)
6          val newHandled = handled ++ newCycle
7          newCycle :: cyclesOf(perm, newHandled)
8        case None =>        // Wenn kein Startwert gefunden wurde
9          Nil
10     }
```

Berechnung der Instruktionen Anhand der berechneten Zyklen wird im nächsten Schritt die Instruktionskette, also die Befehle für die Kranbewegungen, errechnet. Hierfür wird zunächst die folgend dargestellte Methode `computeFromCycles` definiert, welche sich einer - gleich anschließend betrachteten - weiteren Funktion `computeCycle` bedient. Diese Funktion berechnet zunächst mit `computeCycle` die benötigten Instruktionen. Die letzte Instruktion, die von `computeCycle` generiert wird, ist immer ein `PutWag`, dieses ist jedoch überflüssig und wird daher mit `init` gelöscht. Da aber `computeCycle` davon ausgeht, dass vor einem Aufruf bereits ein Container aufgehoben wurde, muss noch ein `TakeCon` vor der Instruktionskette angehängt werden. Außerdem soll eine Liste zurückgegeben werden, weswegen die Instruktionskette zu solch einer umgewandelt wird.

```
1  def computeFromCycles(cycles: Cycles): Seq[Instruction] = {
2    // Lösche letzten Befehl (immer ein PutWag) mit init
3    val (instrs, _) = computeCycle(cycles.head, cycles.tail).init
4    TakeCon :: // Nimm den ersten Container schonmal auf
5      instrs.toList // Wandel den ListBuffer in List um.
6  }
```

Nun wird die Funktion `computeCycle` erläutert. Im Allgemeinen soll diese Funktion 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 und nicht auf der Waggonseite des Gleises transportiert. Das Grundprinzip des Algorithmus' ist es, dass nacheinander alle Elemente des Zyklus' abgearbeitet werden. Dazu wird ein `foldLeft` über den Zyklus ausgeführt (vgl. Z. 17). 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 (vgl. Z. 20). Anschließend werden die durch `foldLeft` erzeugten Instruktionen und restlichen Zyklen zurückgegeben (vgl. Z. 22).

Im folgenden ist zwecks Lesbarkeit der Code dargestellt mit Deklaration aber ohne Definition der Hilfsfunktion `step` (vgl. Z. 7f). Der Funktionsrumpf folgt später. Außerdem wird speziell für die `step` Funktion der type-alias `Step` für das Tuple `(ListBuffer[Instruction], Cycles, Int)` benutzt.

```
def computeCycle(cycle: Cycle, other: Cycles):
  (ListBuffer[Instruction], Cycles) = {
  // Das maximale Element, benutzt von step
  val max = cycle.max

  type Step = (ListBuffer[Instruction], Cycles, Int)
  def step(instrs: ListBuffer[Instruction], cyclesLeft: Cycles,
           prev: Int, cur: Int): Step = [...] // Hier ausgelassen

  val erster = cycle.head
  // Beginne mit leerem ListBuffer
  // den restlichen Zyklen other sowie dem ersten Element
  val initial = (ListBuffer[Instruction](), other, erster)
  // Arbeite alle Elemente des Zyklus' ab
  val (instrs, cyclesLeft, last) =
    // foldLeft über den Zyklus; das erste Element hinten angehängt.
    (cycle.tail :+ erster).foldLeft(initial) {
      case ((instrs, cyclesLeft, prev), cur) =>
        // Die Argumente des foldLeft an step "durchreichen".
        step(instrs, cyclesLeft, prev, cur)
    }
  (instrs, cyclesLeft)
}
```

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

1. Wenn das zuletzt betrachtete Element `prev` das Maximum `max` ist und es einen nächsten Zyklus `nextCycle` gibt, dessen erstes Element `next` eins weiter rechts von `max` bzw. `prev` ist, dann konkateniere die Zyklen entsprechend. Das heißt, es werden 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 ohne restliche 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`. Die möglicherweise "übrig" gebliebenen 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, den aktuellen Container auf den Waggon ablegen und dann den Container auf dem Containerstellplatz aufheben.

Folgend ist der Scala-Code abgebildet, welcher die Hilfsfunktion `step` darstellt. Dieser Codeausschnitt ist deutlich komplexer als die vorherigen. Deswegen wurden entsprechend 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 einen nächsten Zyklus direkt nach diesem beginnend?
7     // ===== (1) =====
8     case Some(nextCycle @ (next :: _)) if prev==max && max+1==next =>
9       // Wenn ja, "konkateniere" diese Zyklen.
10      val (cycleInstrs, _)=computeCycle(cyclesLeft.head,cyclesLeft.tail)
11      val extraInstrs = instrs ++=
12        ListBuffer(PutCon, MoveRight, TakeCon) ++=
13        cycleInstrs ++= ListBuffer(MoveLeft, TakeCon)
14      step(extraInstrs, Nil, prev, cur)
15    // Gibt es einen nächsten Zyklus und beginnt er
16    // vor dem nächsten Element dieses Zyklus'?
17    // ===== (2) =====
18    case Some(nextCycle @ (next :: _)) if cur > next =>
19      // Wenn ja, dann arbeite erst nextCycle ab.
20      val (cycleInstrs, newCyclesLeft) =
21        computeCycle(nextCycle, cyclesLeft.tail)
22      val newInstrs = instrs ++=
23        ListBuffer(Move(prev -> next), Rotate, TakeCon,
24                   Rotate, PutCon, Rotate) ++= cycleInstrs
25      step(newInstrs, newCyclesLeft, next, cur)
26    // ===== (3) =====
27    case _ =>
28      // Andernfalls, fahre einfach mit der Abarbeitung fort.
29      val newInstrs = instrs ++=
30        ListBuffer(Move(prev -> cur), Rotate, PutWag, TakeCon)
31      (newInstrs, cyclesLeft, cur)
32  }
```

2.3.2. 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.

Korrektheit von cycle 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, ersetzte ich die Programmbezeichnungen durch mathematische Bezeichnungen. 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.

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

Da für alle x die im Zyklus ζ enthalten sind, $\zeta(x) = \pi(x)$ gilt und genau die Elemente $i, \zeta(i), \zeta^2(i), \ldots, \zeta^{k-i}(i)$ in ζ enthalten sind, gilt also

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

Nun betrachten wir nochmals die Funktionsweise von `cycle` bzw. von `step`. Wir behaupten zunächst, `step` liefert zu einer Zahl $j = \pi^x(i)$ mit $x \in \{0, \ldots, k\}$ die Zahlen $\pi^x(i), \pi^{x+1}(i), \ldots, \pi^{k-1}(i)$. 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 $j = \pi^x(i)$ mit $x \leq k$ und $x > 0$ bereits die Zahlen $\pi^x(i), \pi^{x+1}(i), \ldots, \pi^{k-1}(i)$. Also zeigen wir nun, dass `step` auch für ein $l = \pi^{x-1}$ die richtigen Zahlen liefert. `step` reiht also l vor die Zahlen, die durch Aufruf von `step` mit $\pi(l) = \pi(\pi^{x-1}) = \pi^x = j$ berechnet werden. Das ergibt genau die Zahlen $\pi^{x-1}(i), \pi^x(i), \pi^{x+1}(i), \ldots, \pi^{k-1}(i)$. 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

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

Korrektheit von cyclesOf 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 und Bezeichner. Die Liste der disjunkten Zyklen, die `cyclesOf` berechnen soll, bezeichne ich mit $\zeta_1, \zeta_2, \ldots, \zeta_c$, 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 $\zeta_1, \zeta_2, \ldots, \zeta_c$ zurückgibt, wobei c die Anzahl disjunkter Zyklen ist und $x < y \Leftrightarrow min(\zeta_x) < min(\zeta_y)$ für alle $x, y = 1 \ldots c$. 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 $Z = \zeta_1 \cup \zeta_2 \cup \cdots \cup \zeta_c$ 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 $x \in \{1, \ldots, c\}$ und $Z = \zeta_1 \cup \zeta_2 \cup \cdots \cup \zeta_{x-1} \cup \zeta_x$ die Zyklen $\zeta_{x+1}, \ldots, \zeta_c$ findet. Wir zeigen, dass dies auch für $x \to x - 1$ gilt. Zunächst wird der Wert $s \in \{1, \ldots, n\}$ ($s =$`start`, n ist die Länge von π) mit $s \notin Z$ gesucht. Nun wird der neue Zyklus $\zeta_{(x-1)+1}$ berechnet. Dieser ist sicher disjunkt von den zuvor berechneten Zyklen, da er bei $s \notin Z$ beginnt. Anschließend wird `cyclesOf` rekursiv aufgerufen, mit `handled` $= \zeta_1 \cup \cdots \cup \zeta_{x-1} \cup \zeta_x$. Dieser Aufruf liefert nach Induktionsannahme die Zyklen $\zeta_{x+1}, \ldots, \zeta_c$. Also werden insgesamt die Zyklen $\zeta_x, \zeta_{x+1}, \ldots, \zeta_c$ ausgegeben. Dass 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 $\zeta_1, \ldots, \zeta_c$ zurückgegeben.

Optimalität des Ergebnisses Anschließend zeigen wir die Optimalität vom eigentlichem Algorithmus, die Berechnung der Kraninstruktionen. Diese machen wir uns klar, indem wir uns erst für eine beliebige Containerkonstellation, also eine beliebige Permutation überlegen, wie ein optimaler Weg aussehen muss.

Äquivalenzklassen Sei π eine beliebige Permutation über $X := [1, n]$ und $Z := \{\zeta_1, \zeta_2, \ldots, \zeta_d\}$ die disjunkten Zyklen, die π darstellen. Nun teilen wir diese Zyklen in die Äquivalenzklassen $A_1, \ldots, A_a$ auf. Die Äquivalenzklassen sollen das Intervall $[1, n]$ in disjunkte Intervalle aufteilen, die zusammen wiederum $[1, n]$ ergeben. Für eine Äquivalenzklasse, die das Intervall $[x, y]$ darstellt, müssen alle Zyklen zwischen x und y liegen. Folgend definieren wir nun unsere Äquivalenzrelation $\equiv$, die diese Eigenschaften erfüllen soll.

Seien $\eta, \theta \in Z$ zwei Zyklen der Länge k bzw. l der Form $e, \eta(e), \ldots, \eta^{k-1}(e)$ bzw. $t, \theta(t), \ldots, \theta^{l-1}(t)$. Weiter seien $E := \{e, \eta(e), \ldots, \eta^{k-1}(e)\}$ und $T := \{t, \theta(t), \ldots, \theta^{l-1}(t)\}$, also jeweils die Mengen der "bewegten" Elemente der Zyklen. Dann gilt die Äquivalenz wie folgt.

$$\eta \equiv \theta \text{ genau dann, wenn } [\min(E) \leq \min(T) \leq \max(E) \vee \min(T) \leq \min(E) \leq \max(T)]^+$$

Hierbei notiert R^+ die transitive Hülle von R. In Worten ausgedrückt, sind zwei Zyklen genau dann äquivalent, wenn sie sich in einem oder mehreren Schritten wechselseitig überlappen.

Nun machen wir uns noch klar, dass $\equiv$ auch wirklich eine Äquivalenzrelation auf Z ist. Die Reflexivität ist einfach, sei $\eta \in Z$ Zyklus und $E :=$ alle Elemente von η (die nicht fest gelassen werden). Dann gilt $\min(E) \leq \min(E) \leq \max(E)$, also gilt $\eta \equiv \eta$.
Die Symmetrie ist ebenfalls recht anschaulich, da $\equiv$ bereits symmetrisch definiert ist.
Die Transitivität folgt aus der Definition als transitive Hülle.
Die Relation $\equiv$ ist also eine Äquivalenzrelation.

Anschließend zeigen wir noch, dass die durch die Äquivalenzrelationen geformten Äquivalenzklassen auch wirklich das Intervall $[1, n]$ in disjunkte Intervalle aufteilen. Seien also $A_1, \ldots, A_a$ die Äquivalenzklassen, in die $\equiv$ die Zyklen aufteilt. Nun setzen wir $\min(A)$ einer Äquivalenzklasse A auf das Minimum der Menge aller Elemente der Zyklen, die A enthält. Genauso setzten wir $\max(A)$ auf das Maximum aller Elemente der Zyklen, die A enthält.

Nun wollen wir uns veranschaulichen, dass für eine Äquivalenzklasse A_x, mit dem Minimum $a := \min(A_x)$ sowie dem Maximum $b := \max(A_x)$ Folgendes gilt: Für jedes $i \in [a, b]$ gibt es genau einen Zyklus $\zeta \in A_x$, der diesen nicht festlässt.
Die Eindeutigkeit folgt aus der Disjunktheit der Zyklen. Dass der Zyklus in A_x enthalten ist, lässt sich wie folgt veranschaulichen. Sei θ der an a beginnende Zyklus und η der an b endende Zyklus. Seien weiter T, E die Mengen der nicht fest gelassenen Elemente von θ, η. Dass $\theta, \eta \in A_x$ ist klar, denn sonst wären die Minima und Maxima von A_x nicht a, b. Nun unterscheiden wir zwischen zwei Fällen.
(1) Falls $a = \min(T) \leq \min(E) \leq \max(T)$, dann muss entweder $\zeta \equiv \theta$ oder $\zeta \equiv \eta$ gelten. Denn falls $i \leq \max(T)$, so gilt aber gleichzeitig $\min(T) \leq i$, also $\zeta \equiv \theta$. Andernfalls gilt $i > \max(T)$, also $\min(E) < i \leq \max(E) = b$. Somit gilt $\zeta \equiv \eta$.
(2) Andernfalls gibt es die Zyklen $\iota_1, \iota_2, \ldots, \iota_j$ mit den zugehörigen Mengen $I_1, I_2, \ldots, I_j$ für die Folgendes gilt.

$$(\min(T) \leq \min(I_1) \leq \max(T)) \wedge (\min(I_1) \leq \min(I_2) \leq \max(I_1)) \wedge \cdots \wedge (\min(I_j) \leq \min(E) \leq \max(I_j))$$

Dass es diese gibt, folgt aus der Definition von $\equiv$ als transitive Hülle. Denn gäbe es diese Zyklen "zwischen" θ und η nicht, so wären diese nicht äquivalent. Nun muss aber i zwischen den Schranken von einem ι_y sein, da alle $\iota_1, \ldots, \iota_j$ zusammengefasst das ganze Intervall $[a, b]$ abdecken. Also muss auch $\zeta \equiv \iota_y$ gelten und somit $\zeta \equiv \theta$.
Wie haben also gezeigt, dass $\zeta \in A_x$.

Schließlich wollen wir zeigen, dass das Intervall $[1, n]$ durch die Äquivalenzklassen in disjunkte Intervalle aufgeteilt wird. Angenommen dies wäre nicht der Fall, so müsste es eine Stelle i geben, die in beiden Äquivalenzklassen enthalten sind. Allerdings gäbe es dann in beiden Äquivalenzklassen einen Zyklus θ bzw. η, der diese Stelle i enthält. Die Zyklen sind jedoch nach Definition disjunkt, also ist dies ein Widerspruch.

Der optimale Weg Was sagen uns jetzt aber die Äquivalenzklassen? Wir erinnern uns, dass diese das Intervall $[1, n]$ in disjunkte Intervalle aufteilen. Gibt es jetzt aber mehr als eine Äquivalenzklasse, heißt das, dass es eben mehrere solche disjunkten Intervalle gibt.

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 geben kann, die sich *nicht* überschneiden! Denn mit diesem Ansatz würden wir nur die erste Äquivalenzklasse abarbeiten, nämlich die, die sich überschneiden.

Deshalb muss das Ende einer Äquivalenzklasse berücksichtigt werden. (Im Code ist dies Fall (1) in der Hilfsmethode `step` von `computeCycle`). Sobald das Ende der Äquivalenzklasse erreicht wird, muss unterbrochen werden, die nächste Äquivalenzklasse abgearbeitet werden und schließlich wieder fortgefahren werden.

Der Algorithmus erkennt das Äquivalenzklassenende korrekt. Es wird immer das bisherige Maximum max aus dem aktuell bearbeiteten Zyklus η gebildet. Beginnt ein Zyklus θ direkt nach diesem Maximum max, so wird unterbrochen. Wie man sich leicht veranschaulicht, ist das die korrekte Unterbrechung, denn wäre θ nicht aus einer anderen Äquivalenzklasse, würde er bereits vorher durch verschachtelte Unterbrechung abgearbeitet worden sein. Es wären also vorher bereits Unterbrechungen der Abarbeitungen erfolgt. Ist dies aber nicht der Fall, gibt es kein Zyklus, der über max hinausreicht und somit sind die Zyklen η und θ mit Sicherheit nicht äquivalent.

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 angelangen 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 Waggonpositionen 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 $o >= x$ zum Waggon $i <= x$ gebracht werden muss. Da jedoch alle Zyklen innerhalb einer 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 denselben Grenzen liegt.

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

a) Der erzeugte Weg ist ein zusammenhängender Weg (keine Sprünge).

b) Jeder Container wird an die richtige Position gebracht.

c) Es werden keine "unnötigen" Fahrten erzeugt.
(Leerfahrten innerhalb einer Äquivalenzklasse, oder Fahrten "in die falsche Richtung").

Dass 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 a) klar. Sei i die jeweilige Position des Krans (im Code `prev`). Dann gilt zu

Beginn $i = 1$. Hier wird mit der ersten Zyklenabarbeitung begonnen. In jedem Schritt wird zwischen den drei Fällen der Hilfsfunktion `step` (2.3.1) unterschieden. Im ersten Fall gilt $i = max$ und der nächste Zyklus beginnt bei $max + 1 = i + 1$. 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 `cur` > `next`. Da `nextCycle` bereits abgearbeitet worden wäre, wenn `prev` ≥ `cur`, gilt `prev` < `cur` < `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 weitergebracht werden. Im dritten Fall ist es trivial, der Container wird von `prev` nach `cur` gebracht, also ist der Weg ebenfalls zusammenhängend.

Nun zeigen wir die Eigenschaft b). Hier unterscheiden wir wieder zwischen den 3 Fällen von `step`. 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 zunächst 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 Nachfolgezyklen, womit der 3. Fall vorliegt. Im zweiten Fall ist die Überlegung ähnlich. Der aktuelle Container wird bei `next` zwischengespeichert, der Kran fährt den Nachfolgezyklus 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. Dass 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 die er positioniert werden muss, ist auch Eigenschaft b) gegeben. Die Eigenschaft c) ist ebenfalls gegeben, denn 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 eines `computeCycle`-Aufrufs) `first=cycle.head` ist, und weil 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 seinen bisherigen Container auf die andere Seite schwenken, den Container $\eta(i)$ aufheben und nochmals 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 haben also bewiesen, dass der Algorithmus - und damit im Groben auch die Implementierung - optimale Ergebnisse liefert.

2.3.3. 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`s), 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 der Annahme, dass wie-

derum 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 zum 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 nichts 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 $a <= n$, 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 Überschneidung wird in Fall 2 danach aufgelöst, sie 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 dieser Fall 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, nicht mehr 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.

Die Funktion `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 Menge der 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

$$O(\sum_{i=0}^{c} O(k(\zeta_i))) = O(O(\sum_{i=0}^{c} k(\zeta_i))) = O(n)$$

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 der c Zyklen gezeigt.

2.4. 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 Worst-case bei $n/2$ Zyklen führt jedoch zu einer Worst-case-Komplexität von $O(n^2)$. Deshalb soll als Erweiterung die Laufzeitkomplexität weiter verringert werden.

Außerdem sind die Algorithmen, wie sie im Abschnitt 2.3 angegeben sind, nicht endrekursiv. 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 10.000 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 4.4 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.

2.4.1. 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 der Scala-Compiler eine Tailrekursionseliminierung vornehmen kann. Das heißt, alle rekursiven Aufrufe einer Funktion müssen der letzte Befehl einer Funktion sein.

Zunächst wurde die Funktion `cycle` auf folgenden Code optimiert. (Die `@tailrec`-Annotation weist daraufhin, dass eine Tailrekursionseliminierung durchgeführt 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(Nil, perm(start - 1)))
8 }
```

Die Änderungen betreffen wesentlich die Methode `step`. Diese hat nun zwei Parameter `ready` und `idx`. In `ready` werden alle bisherig gefundenen Elemente eines Zyklus' akkumuliert. Bei Rekursionsabbruch muss dementsprechend die umgekehrte Liste zurückgegeben werden (vgl. Z. 4), da in einer Liste "last in first out" gilt. Es soll jedoch das zuerst gefundene Element auch als erstes in der Liste stehen. Falls die Abbruchbedingung 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, wie im nachfolgendem Codeausschnitt dargestellt. 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 Parameter `ready` speichert ähnlich wie bei dem neuen verbessertem `cycle` die vor dem Aufruf "gesammelten" Ergebnisse. In diesem Fall werden also die bereits gefundenen Zyklen akkumuliert. Der zweite Parameter ist `prev`. Hier wird entweder der Startwert des vorherigen Zyklus' oder, falls es keinen vorherigen gibt, 0 übergeben. Die Abbruchbedingung bleibt die gleiche, es werden jedoch die akkumulierten Ergebnisse aus `ready` - wieder wie oben bei `cycle` - nach Umkehrung der Reihenfolge zurückgegeben. Die Suche nach dem nächsten Element ist ebenfalls abgeändert. Es wird nicht mehr 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 Nichtenthaltensein 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   }
```

2.4.2. Optimale Ergebnisse

Wie oben (in 2.3.2) 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 immer noch. Insofern liefern auch die neuen Funktionen die gewünschten Zyklen. Die Änderungen wurden oben jeweils so erklärt, dass gleichzeitig die Korrektheit begründet wird.

2.4.3. 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 einen 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 (vgl. Z. 7f. in `cylesOf`) 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` lineare Laufzeitkomplexität vorweisen kann, ist die Gesamtkomplexität $O(n)$.

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

2.4.4. 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

Grob gesehen gliedert sich die Implementierung in folgende Module und Klassen.

Cycler	Algorithmen zur Berechnung der Zyklen (in den zwei Varianten `SlowCycler` und `FastCycler`)
Instructor	Algorithmus zur Berechnung der Kraninstruktionen aus den Zyklen
Gleis	Datenstruktur zur Verwaltung der Containerstellplätze und Waggons
Instructions	Die Instruktionen, die dem Kran mitteilen, was ausgeführt werden muss
Maschine	Klasse zur Simulation einer Maschine
Utils	hilfreiche Methoden
ListBuffer	modifizierte Variante der standardmäßigen Scala-Klasse `ListBuffer`

In den folgenden Abschnitten werden jeweils kurz die Implementierungen erläutert. Der vollständige Programmtext findet sich im entsprechenden Abschnitt des Kapitels 6 “Programmtext”.

3.1. Cycler - Berechnung der Zyklen

Da beide Algorithmen zur Zyklenfindung implementiert werden, ist ein `trait Cycler` implementiert (vgl. Z. 8), welches die einzige Methode des Moduls `cyclesOf(Seq[Int]): Cycles` definiert. Diese soll zu einer gegebenen Permutation eine Liste von nach Startelementen sortierten Zyklen zurückgeben. Die Implementierung des `SlowCycler` (vgl. Z. 13) erfolgte wie in 2.3, die des `FastCycler` (vgl. Z. 34) nach 2.4.

3.2. Instructor - Berechnung der Kraninstruktionen

Im Modul `Instructor` sind Funktionen zur Berechnung der Instruktionen implementiert. Diese gliedern sich in die extern zu benutzenden Funktionen sowie die intern benötigten Hilfsfunktionen. Extern zu verwenden sind `compute(Seq[Int], Cycler): Seq[Instruction]` (vgl. Z. 7) zur Berechnung ausgehend von einer Permutation, sowie `computeFromCycles(Cycles): Seq[Instruction]` (vgl. Z. 9) zur Berechnung der Liste der Instruktionen aus - meist vorher bereits berechneten - Zyklen. Die erstere vereinfacht die Benutzung dadurch, dass nur die Permutation angegeben werden muss (die Zyklen werden dann automatisch berechnet). Die interne Hilfsfunktion, die die eigentliche Berechnung definiert, ist `computeCycle(Cycle, Cycles): (ListBuffer[Instruction], Cycles)` (vgl. Z. 18). 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 Zustands

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 Containers auf einem Waggon. Zu Beginn wird das Array `con` mit der Permutation initialisiert. Falls `con` oder `wag` an einer Position keinen Container enthalten, wird dies durch den Wert 0 dargestellt.

Ein `Gleis` definiert Methoden zum Setzen und Aufheben - also Löschen - von Containern an einem bestimmtem Index. Ein `Gleis` stellt `takeCon(Int): Int` (vgl. Z. 21) zum Löschen auf den Containerstellplätzen bzw. `takeWag(Int): Int` (vgl. Z. 20) zum Löschen auf den Waggons bereit. Die Methoden `putCon((Int, Int)): Int` (vgl. Z. 23) und `putWag((Int, Int)): Int` (vgl. Z. 22) werden zum Setzen von Containern 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.

3.4. Instructions - Die Anweisungen

Es wurden mehrere Objekte implementiert, die die einzelnen Befehle darstellen. Es wurden `TakeWag`, `TakeCon`, `PutWag` und `PutCon` also Lege- und Hebebefehle erstellt, sowie `Rotate` als Rotationsbewegung des Kranes. Diese spielen für den Weg des Kranes keine Rolle. Die Befehle, die den Kranweg unmittelbar beeinflussen, sind `Move` bzw. `MoveLeft` und `MoveRight`. Hierbei ist `Move` die Superklasse der beiden anderen.

3.5. Maschine - Interpretieren der Kraninstruktionen

Um die erzeugten Instruktionen interpretieren, also simuliert ausführen 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.

3.6. Utils - Helfende Methoden

Weitere Methoden, die nützlich im Rahmen des Programms sind, jedoch nicht direkt zur Implementierung der Aufgabenlösung dienen, wurden in das Modul `Utils` ausgelagert. Besondere Bedeutung hat die Funktion `randPerm`, die zu einer gegebenen Permutationslänge eine zufällige Permutation berechnet (vgl. Z. 6-19). Außerdem wurden auch Methoden zum Speichern der Instruktionsketten und Permutationen implementiert.

3.7. ListBuffer - Erweiterung der Standardklasse ListBuffer

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 `ListBuffer`s mit konstantem Zeitaufwand erweitert.

4. Programmabläufe

In diesem Abschnitt sind Programmabläufe dargestellt. Diese sind einerseits Testfälle, die praktische Indizien für die Korrektheit und Optimalität liefern. Andererseits stellen diese auch für sich einen Erkenntnisgewinn dar, insbesondere die Spezialfälle sind hier interessant.

Im Abschnitt 4.1 wird das Beispiel aus der Aufgabenstellung ausgeführt, anschließend werden in 4.2 Spezialfälle diskutiert. Darauf folgend wird in 4.3 mit zufällig erzeugten Permutationen getestet. Zum Schluss wird der in 2.4 entworfene Algorithmus auf Skalierbarkeit in der Praxis getestet (4.4).

Hinweis Die Instruktionen wurden zwecks Platzsparens abgekürzt. Die Abkürzungen sind folgende.

```
TC   TakeCon    TW   TakeWag
PC   PutCon     PW   PutWag
ML   MoveLeft   MR   MoveRight
R    Rotate
```

4.1. Beispiel aus der Aufgabenstellung

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

In der Zeile 15 ist die anhand der Permutation ausgerechnete, mindestens benötigte Weglänge m ausgegeben. In der Zeile 16 ist die tatsächliche Weglänge l ausgegeben.

```
1  scala> val perm = Seq(4,3,2,1)
2  perm: Seq[Int] = List(4, 3, 2, 1)
3
4  scala> val verified = Utils demonstrate perm
5  Time used for computing Cycles: 6
6  The computed Cycles are:
7    (1 4)
8    (2 3)
9  Number of cycles: 2
10 The generated Instructions (shortened) are:
11  [ TC(0) : MR(1) : R(0) : TC(0) : R(0) : PC(0) : R(0) : MR(1)
12  : R(0) : PW(0) : TC(0) : ML(1) : R(0) : PW(0) : TC(0) : MR(2)
13  : R(0) : PW(0) : TC(0) : ML(3) : R(0) : PW(0) ]
14 Time used computing Instructions: 26
15 1 2 3 4;(m=8)
16 4 3 2 1;(l=8)
17 -->      (1)
18   -->    (1)
19   <--    (1)
20   ----> (2)
21 <------ (3)
22 Time used interpreting: 23
23 Verifying results...
24 verified: Boolean = true
```

Bemerkenswert ist hier, dass der erstellte Algorithmus in diesem Fall exakt den gleichen Weg liefert wie im Beispiel der Aufgabenstellung angegeben. Es gibt jedoch noch verschiedene andere optimale 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 verschiedene 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. Spezialfälle

Nachfolgend betrachten wir einige Spezialfälle. Gezeigt werden sowohl ein kleines Beispiel mit 4 Containern, aber auch größere Beispiele mit 14 und 20 Containern. Im nächsten Abschnitt 4.3 folgen weitere große Beispiele.

Äquivalenzklassen

Betrachten wir folgende Permutation $\pi = (21)(43)$. Da sich die zwei Zyklen nicht überschneiden, stellen diese jeweils eine Äquivalenzklasse dar. Es müssen also Befehle erzeugt werden, die diese Grenzen richtig überschreitet. Wir betrachten folgend die Ausgabe des implementierten Programms.

```
scala> Utils demonstrate perm
Time used for computing Cycles: 4
The computed Cycles are:
  (1 2)
  (3 4)
Number of cycles: 2
The generated Instructions (shortened) are:
 [ TC : MR(1) : R : PW : TC : MR(1) : R : TC : R : PC : R : MR(1)
 : R : PW : TC : ML(1) : R : PW : TC : ML(1) : ML(1) : R : PW ]
Time used computing Instructions: 21
1 2 3 4;(m=4)
2 1 4 3;(l=6)
-->     (1)
  -->   (1)
    --> (1)
    <-- (1)
  <--   (1)
<--     (1)
Time used interpreting: 84
Gleis:
Container: _ _ _ _
Waggons:   1 2 3 4
Verifying results...
res7: Boolean = true
```

Betrachtet man alleine die Abbildungen der Indizes auf ihre Bilder, so erkennt man, dass insgesamt 4 Weglängen gefahren werden müssen, dies ist der Wert m. Allerdings sind auch zwei zusätzliche Weglängen zu fahren, denn der Kran kann sonst nicht alle Container erreichen und auf den Waggon heben. Addiert man diese zwei Weglängen zu den anderen vier, so erhält man sechs. Der berechnete Weg - hier der Wert l - ist also optimal.

Zyklen in Zyklen in Zyklen

Nun betrachten wir ein maximal verschachteltes Beispiel. Es werden immer zwei Zyklen in einen größeren verschachtelt. Bei einer Permutationslänge von 14 ist also $\pi = (1\ 14)(2\ 7)(3\ 4)(5\ 6)(8\ 13)(9\ 10)(11\ 12)$. Der Zyklus (1 14) enhält (2 7) sowie (8 13), diese wiederum jeweils zwei weitere Zyklen. Es wird also der Baum der Zyklenschachtelung als Folge dargestellt.

Nachfolgend ist die Ausführung des Beispiels angegeben.

```
scala> val perm = Seq(14, 7, 4, 3, 6, 5, 2, 13, 10, 9, 12, 11, 8, 1)
perm: Seq[Int] = List(14, 7, 4, 3, 6, 5, 2, 13, 10, 9, 12, 11, 8, 1)

scala> val verified = Utils demonstrate perm
```

```
Time used for computing Cycles: 0
The computed Cycles are:
  (1 14)
   (2 7)
    (3 4)
    (5 6)
   (8 13)
    (9 10)
    (11 12)
Number of cycles: 7
The generated Instructions (shortened) are:
 [ TC : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : PW : TC : ML(1) : R : PW : TC : MR(2) : R : TC
 : R : PC : R : MR(1) : R : PW : TC : ML(1) : R : PW : TC : MR(2)
 : R : PW : TC : ML(5) : R : PW : TC : MR(6) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R : MR(1) : R : PW : TC : ML(1)
 : R : PW : TC : MR(2) : R : TC : R : PC : R : MR(1) : R : PW
 : TC : ML(1) : R : PW : TC : MR(2) : R : PW : TC : ML(5) : R : PW
 : TC : MR(6) : R : PW : TC : ML(13) : R : PW ]
Time used computing Instructions: 0
 1  2  3  4  5  6  7  8  9 10 11 12 13 14;(m=54)
14  7  4  3  6  5  2 13 10  9 12 11  8  1;(l=54)
--->                                      (1)
   --->                                   (1)
      --->                                (1)
      <---                                (1)
      ------>                             (2)
            --->                          (1)
            <---                          (1)
            ------>                       (2)
   <---------------                       (5)
   ------------------>                    (6)
                      --->                (1)
                         --->             (1)
                         <---             (1)
                         ------>          (2)
                               --->       (1)
                               <---       (1)
                               ------>    (2)
                      <---------------    (5)
                      ------------------> (6)
<------------------------------------- (13)
Time used interpreting: 2
Gleis:
Container: _ _ _ _ _ _ _ _ _ _  _  _  _  _  _
Waggons:   1 2 3 4 5 6 7 8 9 10 11 12 13 14
Verifying results...
verified: Boolean = true
```

Wie man sieht, ist $m = l = 54$. Dies kommt daher, weil wir unsere Zyklen bewusst als Schachtelung konstruiert haben. Gut zu sehen ist auch, wie der Kran sich schrittweise eine Ebene tiefer in die Zyklenschachtelung arbeitet. Bevor ein Zyklus vollständig abgearbeitet werden kann, müssen alle Zyklen innerhalb von diesen bereits abgearbeitet sein.

Identität

Ein weiterer Spezialfall ist die Identität als Permutation. Das heißt $\pi(x) = x$ für alle $x \in \{1, \ldots, n\}$. Hierbei bildet jeder Container eine eigene Äquivalenzklasse, da keine überschneidenden Zyklen existieren. Nachfolgend wird also die nötigte Kranfahrt für 20 bereits in der richtigen Reihenfolge auf den Containerstellplätzen liegenden Container simuliert.

```
scala> val perm = (1 to 20) // Identität erzeugen
perm: scala.collection.immutable.Range.Inclusive
 with scala.collection.immutable.Range.ByOne =
  Range( 1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
        11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

scala> val verified = Utils demonstrate perm
Time used for computing Cycles: 0
The computed Cycles are:
  (1)
  (2)
  (3)
  (4)
  (5)
  (6)
  (7)
  (8)
  (9)
  (10)
  (11)
  (12)
  (13)
  (14)
  (15)
  (16)
  (17)
  (18)
  (19)
  (20)
Number of cycles: 20
The generated Instructions (shortened) are:
 [TC : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R : MR(1) : R : TC : R : PC
 : R : MR(1) : R : TC : R : PC : R
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW : TC : ML(1)
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW : TC : ML(1)
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW : TC : ML(1)
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW : TC : ML(1)
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW : TC : ML(1)
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW : TC : ML(1)
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW : TC : ML(1)
```

```
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW : TC : ML(1)
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW : TC : ML(1)
 : ML(0) : R : PW : TC : ML(1) : ML(0) : R : PW ]
Time used computing Instructions: 1
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20;(m=0)
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20;(l=38)
--->                                                        (1)
   --->                                                     (1)
      --->                                                  (1)
         --->                                               (1)
            --->                                            (1)
               --->                                         (1)
                  --->                                      (1)
                     --->                                   (1)
                        --->                                (1)
                           --->                             (1)
                              --->                          (1)
                                 --->                       (1)
                                    --->                    (1)
                                       --->                 (1)
                                          --->              (1)
                                             --->           (1)
                                                --->        (1)
                                                   --->     (1)
                                                      --->  (1)
                                                         <  (0)
                                                      <---  (1)
                                                      <     (0)
                                                   <---     (1)
                                                   <        (0)
                                                <---        (1)
                                                <           (0)
                                             <---           (1)
                                             <              (0)
                                          <---               (1)
                                          <                  (0)
                                       <---                  (1)
                                       <                     (0)
                                    <---                     (1)
                                    <                        (0)
                                 <---                        (1)
                                 <                           (0)
                              <---                           (1)
                              <                              (0)
                           <---                              (1)
                           <                                 (0)
                        <---                                 (1)
                        <                                    (0)
                     <---                                    (1)
                     <                                       (0)
                  <---                                       (1)
                  <                                          (0)
               <---                                          (1)
               <                                             (0)
            <---                                             (1)
            <                                                (0)
         <---                                                (1)
         <                                                   (0)
      <---                                                   (1)
      <                                                      (0)
   <---                                                      (1)
   <                                                         (0)
```

```
103              <---                                                (1)
104              <                                                   (0)
105           <---                                                   (1)
106           <                                                      (0)
107        <---                                                      (1)
108        <                                                         (0)
109     <---                                                         (1)
110     <                                                            (0)
111  <---                                                            (1)
112  <                                                               (0)
113  Time used interpreting: 3
114  Gleis:
115  Container:  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _
116  Waggons:    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
117  Verifying results...
118  verified: Boolean = true
```

Hier ist zwar $m = 0$, denn es muss kein einziger Container nach links oder rechts bewegt werden. Allerdings gibt es $20 - 1 = 19$ Äquivalenzklassen, die "konkateniert" werden müssen. In dem Optimalitätsbeweis haben wir bewiesen, dass der minimale Weg $w = a \cdot 2 + \sum_{i=1}^{n} |i - \pi(i)|$ ist. Da $a = 20 - 1 = 19$ und $\sum_{i=1}^{n} |i - \pi(i)| = 0$, ist $w = 19 \cdot 2 = 38$. Also ist auch der tatsächlich gefahrene Weg $l = w$, also optimal.

Außerdem bemerkenswert ist, dass die Weglänge von 38 für *jede* Permutation der Länge 20 zurückgelegt werden muss. Denn der Kran muss auf jeden Fall bis zum letzten Container und wieder zurück an die Startposition, also $2 \cdot (n - 1)$ Weglängen zurücklegen, wobei n die Länge der Permutation ist.

4.3. Zufällig erzeugte Permutationen

1. Beispiel

Ein nächstes, etwas größeres Beispiel ergibt sich aus zufälliger Erzeugung einer Permutation der Länge 20. Die Permutation selber ist in Zeile 24 und 25 ausgegeben.

```
1  scala> import de.voodle.tim.bwinf.container._
2  import de.voodle.tim.bwinf.container._
3
4  scala> val verified = Utils.demonstrate(20, print = true)
5  Time used for computing Cycles: 8
6  The computed Cycles are:
7    (1 5 6)
8    (2)
9    (3 14 15 10 8 12 19 16 17 20)
10   (4 9 11 18 7)
11   (13)
12 Number of cycles: 5
13 The generated Instructions (shortened) are:
14  [ TC : MR(1) : R : TC : R : PC : R : ML(0) : R : PW : TC : MR(1)
15  : R : TC : R : PC : R : MR(1) : R : TC : R : PC : R : MR(5)
16  : R : PW : TC : MR(2) : R : PW : TC : MR(2) : R : TC : R : PC
17  : R : ML(0) : R : PW : TC : MR(5) : R : PW : TC : ML(11) : R : PW
18  : TC : ML(3) : R : PW : TC : MR(10) : R : PW : TC : MR(1) : R : PW
19  : TC : ML(5) : R : PW : TC : ML(2) : R : PW : TC : MR(4) : R : PW
20  : TC : MR(7) : R : PW : TC : ML(3) : R : PW : TC : MR(1) : R : PW
21  : TC : MR(3) : R : PW : TC : ML(17) : R : PW : TC : MR(2) : R : PW
22  : TC : MR(1) : R : PW : TC : ML(5) : R : PW ]
23 Time used computing Instructions: 36
24  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20;(m=92)
25  5  2 14  9  6  1  4 12 11  8 18 19 13 15 10 17 20  7 16  3;(l=92)
26 --->                                                        (1)
27    <                                                        (0)
28    --->                                                     (1)
29       --->                                                  (1)
30          ---------------->                                  (5)
31                          ------>                            (2)
32                                ------>                      (2)
33                                      <                      (0)
34                                      ---------------->      (5)
35                   <-----------------------------------      (11)
36          <----------                                        (3)
37          ------------------------------>                    (10)
38                                        --->                 (1)
39                             <----------------               (5)
40                      <------                                (2)
41                      ------------>                          (4)
42                                   ---------------------->   (7)
43                                               <----------   (3)
44                                               --->          (1)
45                                                  ---------> (3)
46       <-------------------------------------------------    (17)
47       ------>                                               (2)
48             --->                                            (1)
```

```
<---------------                                                    (5)
Time used interpreting: 30
Gleis:
Container:  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _
Waggons:    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
Verifying results...
verified: Boolean = true
```

Permutationen bis zu einer Länge von 20 können wie gezeigt problemlos in der Konsole angezeigt und dargestellt werden. Durch das gewählte - an die Aufgabenstellung angelehnte - Ausgabeformat können auch die zu fahrenden Wege gut in der Konsole dargestellt werden. Auch die Optimalität des Weges kann leicht nachvollzogen werden. Denn die anhand der Permutation ausgerechnete Weglänge m stimmt mit der tatsächlichen Weglänge l überein.

2. Beispiel

Folgend ist nun noch die Bearbeitung einer weiteren zufälligen Permutation der Länge 20 dargestellt.

```
scala> val verified = Utils.demonstrate(20, print = true)
Permutation (shortened) is:
  (3 11 13 6 9 14 10 7 4 1 19 12 15 5 16 17 18 20 2 8)
Time used for computing Cycles: 0
The computed Cycles are:
  (1 3 13 15 16 17 18 20 8 7 10)
  (2 11 19)
  (4 6 14 5 9)
  (12)
Number of cycles: 4
The generated Instructions (shortened) are:
 [ TC : MR(1) : R : TC : R : PC : R : MR(2) : R : TC : R : PC
 : R : MR(2) : R : PW : TC : MR(6) : R : TC : R : PC : R : ML(0)
 : R : PW : TC : MR(2) : R : PW : TC : ML(9) : R : PW : TC : MR(4)
 : R : PW : TC : ML(5) : R : PW : TC : MR(7) : R : PW : TC : MR(8)
 : R : PW : TC : ML(17) : R : PW : TC : MR(1) : R : PW : TC : MR(10)
 : R : PW : TC : MR(2) : R : PW : TC : MR(1) : R : PW : TC : MR(1)
 : R : PW : TC : MR(1) : R : PW : TC : MR(2) : R : PW : TC : ML(12)
 : R : PW : TC : ML(1) : R : PW : TC : MR(3) : R : PW : TC : ML(9)
 : R : PW ]
Time used computing Instructions: 0
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20;(m=106)
 3 11 13  6  9 14 10  7  4  1 19 12 15  5 16 17 18 20  2  8;(l=106)
--->                                                               (1)
   ------>                                                         (2)
         ------>                                                   (2)
               ------------------>                                 (6)
                                  <                                (0)
                                  ------>                          (2)
            <----------------------------                          (9)
            ------------>                                          (4)
         <---------------                                          (5)
         ---------------------->                                   (7)
                                ------------------------>          (8)
   <-------------------------------------------------------       (17)
   --->                                                            (1)
      ------------------------------>                              (10)
```

```
38                                        ------>                 (2)
39                                              --->              (1)
40                                                 --->           (1)
41                                                    --->        (1)
42                                                       ------>  (2)
43                        <-------------------------------------  (12)
44                    <---                                        (1)
45                    --------->                                  (3)
46 <--------------------------                                    (9)
47 Time used interpreting: 2
48 Gleis:
49 Container:  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _
50 Waggons:    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
51 Verifying results...
52 verified: Boolean = true
```

Auch hier ist die Optimalität gegeben, wie man an $m = l$ sieht. Interessant zu beobachten ist, dass die minimale Weglänge nicht konstant für eine Permutationslänge ist. Obwohl beide Permutationen die Länge 20 haben, müssen in diesem 2. Beispiel längere Wege zurückgelegt werden.

4.4. Demonstration der Skalierbarkeit

Nun soll die Skalierbarkeit demonstriert werden, die als Erweiterung in Form von endrekursiven Funktionen und linearer Laufzeitkomplexität implementiert wurde.
Hierfür erzeugen wir eine zufällige Permutation mit einer Länge von 2^{22} (etwa 4,2 Millionen), die unsere Container darstellt. Längere Permutation sind in der Realität nicht zu erwarten, denn wenn jeder Container nach Standard 40 Fuß lang ist, ergibt dies eine Gesamtlänge g des Gleises von

$$g = 40 \cdot 2^{22} \text{ Fuß} = 167.772.160 \text{ Fuß} \approx 51.137 \text{ km}.$$

Dies ist bereits deutlich länger als der Umfang der Erde am Äquator. 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.

```
scala> val verified = Utils.demonstrate(math.pow(2,22).toInt, false)
Time used for computing Cycles: 3784
Number of cycles: 17
Time used computing Instructions: 22682
(m=5863759046970)
(l=5863759046970)
Time used interpreting: 46230
Verifying results...
verified: Boolean = true
```

Interessant ist hier die Beobachtung, dass es nur 17 Zyklen und eine Äquivalenzklasse gibt, bei einer Permutationslänge von 2^{22}. Insgesamt wurden 22.682 Millisekunden, also gute 22 Sekunden bzw. gut eine Drittel-Minute benötigt, um die Instruktionen zu berechnen. Dies ist ein praktischer Beleg der oben theoretisch bewiesenen guten Laufzeitkomplexität. Nach der Berechnung der Instruktionen wurden diese testweise interpretiert. Hierfür wurden gute 46 Sekunden benötigt. Zum Schluss wurde außerdem verifiziert, dass jeder Container auf der richtigen Position ist. Auch dieses Beispiel liefert einen optimalen Weg, wie man an $m = l = 5863759046970$ sieht. Dieser Weg wäre in der Realität nur schwer zu erreichen, denn nimmt man an, dass der Kran keine Zeit für das Aufheben und Aufnehmen von Containern braucht, sowie pro Sekunde eine Weglänge zurücklegen kann (bei 40 Fuß Containerlänge wären das etwa 44 km/h), so ergibt sich folgende Rechnung.

$$l\ s = 5863759046970/3600\ h = 1628822000/24\ d = 67867583/365,25\ a = 185811,32\ a \approx 1,8 \cdot 10^5\ a$$

Es müsste also über *hundert-achtzig-tausend* Jahre gefahren werden!

5. Programmnutzung

Die Nutzung des Programms erfolgt primär über eine Scala-Console mit richtig eingestelltem Classpath. Um dies einfach zu erreichen, empfehle ich, die Scala-Console nach Anleitung im Kapitel “Allgemeines” (A. 5) zu starten. Die notwendigen Module und Klassen wurden bereits importiert, sofern nach der oben referenzierten Anleitung vorgegangen wurde. Nachfolgend ist immer der Befehl hinter `scala>` angegeben. In den darauf folgenden Zeilen ist die Ausgabe dargestellt.

5.1. Permutationen erzeugen

Permutationen erzeugt man entweder durch direkte Eingabe oder man lässt eine randomisierte Permutation für eine gegebene Länge erzeugen.

Um eine Permutation direkt einzugeben, kann man einfach die Hilfsfunktionen der Scala-Bibliothek benutzen. Man erzeugt und speichert einfach das Bild der Permutation in einer `Seq`. Die Permutation aus der Aufgabenstellung gibt man beispielsweise wie folgt ein.

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

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

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

5.2. Kraninstruktionen erzeugen

Nachdem nun eine Permutation erzeugt wurde, kann die Methode `compute` des Moduls `Instructor` verwendet werden, um die Kraninstruktionen zu berechnen.

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

Es können auch - wenn man will - zunächst die Zyklen berechnet werden, mit dem schnellerem `FastCycler` oder mit dem langsameren `SlowCycler`. Hierzu wird einfach die Methode `cyclesOf` aufgerufen. Zum Beispiel wie nachfolgend angegeben.

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

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

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

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

5.3. Simulation der Kranmaschine

Nun wurde bereits eine Instruktionskette `instrs` erzeugt. Die in der Aufgabenstellung skizzierte Maschine wurde implementiert, um die Kraninstruktionen ausführen zu können. Es kann natürlich auch ein eigener Interpreter geschrieben werden, der die Instruktionen interpretiert. Deswegen wurde in der Klasse `Maschine` extra die Methode `act(instr: Option[Instruction]): Unit` implementiert, so dass eine Unterklasse Extrafunktionen bereitstellen kann.

Am einfachsten ist es, eine `Maschine` zu erzeugen, diese die Instruktionen ausführen zu lassen und anschließend die Ausgabe zu betrachten.

Erzeugen der `Maschine`:

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

Ausführen der Instruktionen:

```
scala> maschine interpret instrs
1 2 3 4;(m=8)
4 3 2 1;(l=8)
-->     (1)
  -->   (1)
  <--   (1)
  ----> (2)
<------ (3)
res1: de.voodle.tim.bwinf.container.Gleis =
Container: _ _ _ _
Waggons:   1 2 3 4
```

In Scala liefert jeder Ausdruck einen Wert zurück. Da in dem obigem Aufruf der Wert jedoch keiner Variable explizit zugewiesen wird, wird der Wert einer generierten Variable `resX` zugewiesen, wobei `X` mit jedem solchen Aufruf inkrementiert wird.

5.4. Zeitmessung

Wenn die Skalierbarkeit nachvollzogen werden soll, empfiehlt sich die Funktion `demonstrate` im Modul `Utils` auszuprobieren. Um beispielsweise für 100.000 Container Instruktionen ausführen zu lassen und anschließend verifizieren zu lassen, ob auch jeder Container am richtigen Platz angekommen ist, führt man folgende Befehle aus. Die Zeitangaben sind jeweils in Millisekunden.

```
scala> val verified = Utils.demonstrate(100000, print = false)
Time used for computing Cycles: 54
Number of cycles: 20
Time used computing Instructions: 343
(m=3338582674)
(l=3338582674)
Time used interpreting: 469
Verifying results...
verified: Boolean = true
```

6. Programmtext

Alle Quelldateien dieser Aufgabe finden sich auf der CD unter `Aufgabe2/src/`.

6.1. Cycler

```scala
package de.voodle.tim.bwinf.container
import annotation.tailrec

object Cycler {
  type Cycle = List[Int]
  type Cycles = List[List[Int]]
}
trait Cycler extends Function1[Seq[Int], List[List[Int]]] {
  def apply(perm: Seq[Int]) = cyclesOf(perm)
  def cyclesOf(perm: Seq[Int]): List[List[Int]]
}
import Cycler._
object SlowCycler extends Cycler {
  def cycle(perm: Seq[Int], start: Int): Cycle = {
    def step(idx: Int): Cycle = // Hilfsfunktion
      if(start == idx)
        Nil
      else
        idx :: step(perm(idx - 1))
    start :: step(perm(start-1))
  }
  def cyclesOf(perm: Seq[Int]): Cycles = cyclesOf(perm, Set())
  def cyclesOf(perm: Seq[Int], handled: Set[Int]): Cycles =
    // up to n calls; accessing Hashset O(1)
    (1 to perm.length) find (i => !handled.contains(i)) match {
      case Some(start) =>
        val newCycle = cycle(perm, start)
        val newReady = handled ++ newCycle // O(n)
        newCycle :: cyclesOf(perm, newReady)
      case None =>
        Nil
    }
}
object FastCycler extends Cycler {
  def cyclesOf(perm: Seq[Int]): Cycles =
    cyclesOf(Nil, perm, new Array[Boolean](perm.length), 0)

  @tailrec private
  def cyclesOf(ready: List[Cycle], perm: Seq[Int],
               handled: Array[Boolean], prev: Int): Cycles = // c *
    (prev+1 to perm.length) find (i => !(handled(i-1))) match { // O(i_c)
      case Some(start) =>
        val newCycle = cycle(perm, start)
        for (i <- newCycle) handled(i-1) = true
        cyclesOf(newCycle :: ready, perm, handled, start)
      case None =>
        ready.reverse // O(n)
    }
  /** Small helper function, finding one cycle. */
  private def cycle(perm: Seq[Int], start: Int): Cycle = {// O(n_c)
    @tailrec def step(ready: List[Int], idx: Int): Cycle = // O(n_c)
               if(start == idx) ready.reverse // O(1)
               else step(idx :: ready, perm(idx - 1))
    (start :: step(Nil, perm(start - 1)))
  }
}
```

6.2. Instructor

```scala
package de.voodle.tim.bwinf.container
import scala.annotation.tailrec
import scala.collection.mutable.tim.ListBuffer
import Cycler._ // import types.

object Instructor {
  def compute(perm: Seq[Int], cycler: Cycler = FastCycler): Seq[Instruction] =
    computeFromCycles(cycler cyclesOf perm)
  def computeFromCycles(cycles: Cycles): Seq[Instruction] =
    TakeCon :: computeCycle(cycles.head, cycles.tail, 1)._1.init.toList

  /**
   * Should be called, after a TakeCon!
   * When a cycle starts, all the containers in the cycles are supposed
   * to be on the container side.
   * Container are always transported on the Container side!
   */
  private def computeCycle(cycle: Cycle, other: Cycles,
                           prevMax: Int): (ListBuffer[Instruction], Cycles) = {
    // Where is the bound of this equivalence class?
    val max = math.max(cycle.max, prevMax)

    type Step = (ListBuffer[Instruction], Cycles, Int)
    @tailrec def step(instrs: ListBuffer[Instruction], cyclesLeft: Cycles,
                      prev: Int, cur: Int): Step =
     cyclesLeft.headOption match {
       case Some(nextCycle @ (next :: _)) if prev==max && max+1==next =>//(1)
         val (cycleInstrs, _) = computeCycle(nextCycle, cyclesLeft.tail, max)
         val extraInstrs = instrs ++=
           ListBuffer(MoveRight, Rotate, TakeCon, Rotate, PutCon, Rotate) ++=
           cycleInstrs += MoveLeft
         step(extraInstrs, Nil, prev, cur)
       case Some(nextCycle @ (next :: _)) if next < cur => // (2)
         val (cycleInstrs, newCyclesLeft) =
           computeCycle(nextCycle, cyclesLeft.tail, max)
         // Move from prev to nextCycle.head (next)
         val newInstrs = instrs ++=
           ListBuffer(Move(prev -> next), Rotate, TakeCon,
                      Rotate, PutCon, Rotate) ++=
           cycleInstrs
         step(newInstrs, newCyclesLeft, next, cur)
       case _ => // (3)
         val newInstrs = instrs ++=
           ListBuffer(Move(prev -> cur), Rotate, PutWag, TakeCon)
         (newInstrs, cyclesLeft, cur)
     }

    val erster = cycle.head
    val initial = (ListBuffer[Instruction](), other, erster)
    val (instrs, cyclesLeft, last) = (initial /: (cycle.tail :+ erster)) {
        case ((instrs, cyclesLeft, prev), cur) =>
            step(instrs, cyclesLeft, prev, cur)
      }
    (instrs, cyclesLeft)
  }
}
```

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,
16               "arr(i-1) at " + i + " must be 0, but is " + arr(i-1))
17       arr(i-1) = what
18   }
19
20   def takeWag(i: Int) = arrTake(wag)(i)
21   def takeCon(i: Int) = arrTake(con)(i)
22   def putWag(map: (Int, Int)) = arrPut(wag)(map)
23   def putCon(map: (Int, Int)) = arrPut(con)(map)
24
25   private def arrString(arr: Array[Int]) = // Only print first 100
26     arr take 100 map (i => if(i == 0) "_" else i.toString) mkString " "
27   override def toString =
28      "Container: " + arrString(con) + "\n" +
29      "Waggons:   " + arrString(wag)
30
31   // Immutable Vector copies!
32   def container = Vector(con: _*)
33   def waggons = Vector(wag: _*)
34 }
```

6.4. Instructions

```
1  package de.voodle.tim.bwinf.container
2
3  sealed trait Instruction {
4    def len: Int = 0
5    def short: String = (toString filter (_.isUpper))
6  }
7  case object TakeWag extends Instruction
8  case object TakeCon extends Instruction
9  case object PutWag extends Instruction
10 case object PutCon extends Instruction
11 case object Rotate extends Instruction
12 sealed trait Move extends Instruction {
13   override def short: String = (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 case class  MoveRight(override val len: Int) extends Move
25 object MoveLeft  extends MoveLeft(1)
26 object MoveRight extends MoveRight(1)
```

6.5. 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) => BigInt(((i+1)-v).abs) } sum
14
15   def log(str: =>Any)(implicit forcePrint: Option[String] = None) =
16     forcePrint match {
17       case Some(force) if print => println(str + force)
18       case Some(force) => println(force)
19       case None if print => println(str)
20       case _ => ()
21     }
22
23   def logInts(ints: =>Seq[Int]): String =
24     (for(i <- ints) yield {
25         val diff = numLength - digits(i)
26         "␣" * diff + i
27       }) mkString ("␣")
28
29   def interpret(instrs: Seq[Instruction]): Gleis = {
30     log(logInts(1 to length) + ";")(Some("(m=" + minLength + ")"))
31     log(logInts(gleis.container) + ";")
32        (Some("(l=" + instrs.map(i => BigInt(i.len)).sum + ")"))
33     interpret(instrs.toList,0,0,1)
34   }
35   // Attach point for further actions (for subclasses)
36   protected def act(instrs: Option[Instruction]) {}
37
38   @tailrec private def interpret(instrs: List[Instruction],
39                                  con: Int, wag: Int, idx: Int): Gleis = {
40     act(instrs.headOption)
41     instrs match { // Recursivly check
42       case Rotate :: xs =>
43         interpret(xs,wag,con,idx)
44       case TakeCon :: xs =>
45         interpret(xs, gleis takeCon idx, wag, idx)
46       case TakeWag :: xs =>
47         interpret(xs, 0, gleis takeWag idx, idx)
48       case PutCon :: xs =>
49         gleis putCon (idx -> con)
50         interpret(xs, 0, wag, idx)
51       case PutWag :: xs =>
52         gleis putWag (idx -> wag)
53         interpret(xs, con, 0, idx)
54       case MoveRight(len) :: xs =>
55         log(space * (idx-1) + arrow * len + ">" +
56             space * (length-len-idx) + "␣(" + len + ")")
57         interpret(xs, con, wag, idx+len)
```

```
58        case MoveLeft(len) :: xs =>
59          log(space * (idx-1-len) + "<" + arrow * len +
60              space * (length-idx) + "␣(" + len + ")")
61          interpret(xs, con, wag, idx-len)
62        case Nil => gleis // Do Nothing
63      }
64    }
65    override def toString = gleis.toString
66  }
67  object Maschine {
68    private def digits(num: Int) = (math.log10(num) + 1).floor.toInt
69  }
```

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 // O(n)
10     for (idx <- 0 until n) a(idx) = idx + 1
11     // randomize array // O(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(perm: Seq[Int], print: Boolean = true) = {
22     val startTime = System.currentTimeMillis
23     val cycles = FastCycler cyclesOf perm
24     println("Time␣used␣for␣computing␣Cycles:␣" +
25             (System.currentTimeMillis - startTime))
26     if(print) {
27       println("The␣computed␣Cycles␣are:␣")
28       println(cycles map (_.mkString("(", "␣", ")")) mkString ("␣␣", "\n␣␣", ""))
29     }
30     println("Number␣of␣cycles:␣" + cycles.length)
31     val instrs = Instructor computeFromCycles cycles
32     val endTime = System.currentTimeMillis
33     if(print) {
34       println("The␣generated␣Instructions␣(shortened)␣are:␣")
35       val instrsGroups = instrs.view.take(20*12).map(_.short).grouped(12)
36       println(instrsGroups.map(_.mkString("␣:␣"))
37               .mkString("␣[␣", "\n␣:␣",
38                         if(instrs.lengthCompare(20*12)>0) "␣..." else "␣]␣"))
39     }
40     println("Time␣used␣computing␣Instructions:␣" + (endTime - startTime))
41     val gleis = new Gleis(perm)
42     val maschine = new Maschine(gleis, print)
43     maschine interpret instrs
44     println("Time␣used␣interpreting:␣" + (System.currentTimeMillis - endTime))
45     if(print) {
46       println("Gleis:␣")
47       println(gleis)
48     }
49     println("Verifying␣results...")
50     gleis.waggons.zipWithIndex forall (xy => xy._1 == xy._2 + 1)
51   }
52
53   def demonstrate(n: Int, print: Boolean): Boolean = {
54     val perm = randPerm(n)
55     if(print) {
56       println("Permutation␣(shortened)␣is:")
57       println(perm.take(480).mkString(
```

```
58            "␣␣␣(", "␣", if(perm.lengthCompare(480) > 1) "␣..." else ")"))
59      }
60      demonstrate(perm, print)
61    }
62  }
```

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   *  @author  Tim Taubner
10  *  @author  Matthias Zenger
11  *  @author  Martin Odersky
12  *  @version 2.8.tim
13  *  [...]
14  */
15 @serializable @SerialVersionUID(34196396135358366 1L)
16 final class ListBuffer[A]
17       extends Buffer[A]
18          with GenericTraversableTemplate[A, ListBuffer]
19          with BufferLike[A, ListBuffer[A]]
20          with Builder[A, List[A]]
21          with SeqForwarder[A]
22 {
23   override def companion: GenericCompanion[ListBuffer] = ListBuffer
24
25   import scala.collection.Traversable
26
27   private var start: List[A] = Nil
28   private var last0: ::[A] = _
29   private var exported: Boolean = false
30   private var len = 0
31
32   protected def underlying: immutable.Seq[A] = start
33
34   /** The current length of the buffer.
35    *  This operation takes constant time.
36    */
37   override def length = len
38
39   // Implementations of abstract methods in Buffer
40
41   override def apply(n: Int): A =
42     if (n < 0 || n >= len) throw new IndexOutOfBoundsException(n.toString())
43     else super.apply(n)
44
45   /** Replaces element at index `n` with the new element
46    *  `newelem`. Takes time linear in the buffer size. (except the
47    *  first element, which is updated in constant time).
48    *  @param n  the index of the element to replace.
49    *  @param x  the new element.
50    *  @throws Predef.IndexOutOfBoundsException if `n` is out of bounds.
51    */
52   def update(n: Int, x: A) {
53     try {
54       if (exported) copy()
55       if (n == 0) {
56         val newElem = new :: (x, start.tail);
57         if (last0 eq start) {
```

```
58            last0 = newElem
59          }
60          start = newElem
61        } else {
62          var cursor = start
63          var i = 1
64          while (i < n) {
65            cursor = cursor.tail
66            i += 1
67          }
68          val newElem = new :: (x, cursor.tail.tail)
69          if (last0 eq cursor.tail) {
70            last0 = newElem
71          }
72          cursor.asInstanceOf[::[A]].tl = newElem
73        }
74      } catch {
75        case ex: Exception => throw new IndexOutOfBoundsException(n.toString())
76      }
77    }
78
79    // THIS PART IS NEW (by tim8dev):
80
81    /** Appends a single element to this buffer.
82     *  This operation takes constant time.
83     *  @param x  the element to append.
84     *  @return   this $coll.
85     */
86    def += (x: A): this.type = {
87      val newLast = new :: (x,Nil)
88      append(newLast, newLast, 1)
89    }
90
91    override def ++=(xs: TraversableOnce[A]): this.type = xs match {
92      case some : ::[A] =>
93        append(some, some.last.asInstanceOf[::[A]], some.length)
94      case buff : ListBuffer[A] =>
95        buff.start match {
96          case some : ::[A] =>
97            if(buff.exported)
98              buff.copy()
99            buff.exported = true
100           append(some, buff.last0, buff.len)
101         case Nil =>
102           this
103       }
104     case xs =>
105       super.++=(xs)
106   }
107
108   private def append(x: ::[A], last: ::[A], length: Int): this.type = {
109     if(exported) copy()
110     if(start.isEmpty) {
111       last0 = last
112       start = x
113     } else {
114       val last1 = last0
115       last1.tl = x
116       last0 = last
```

```
117     }
118     len += length
119     this
120   }
121
122   // END OF NEW PART (by tim8dev).
123
124   /** Clears the buffer contents.
125    */
126   def clear() {
127     start = Nil
128     exported = false
129     len = 0
130   }
131
132   /** Prepends a single element to this buffer. This operation takes constant
133    *  time.
134    *  @param x  the element to prepend.
135    *  @return   this $coll.
136    */
137   def +=: (x: A): this.type = {
138     if (exported) copy()
139     val newElem = new :: (x, start)
140     if (start.isEmpty) last0 = newElem
141     start = newElem
142     len += 1
143     this
144   }
145
146   /** Inserts new elements at the index `n`. Opposed to method
147    *  `update`, this method will not replace an element with a new
148    *  one. Instead, it will insert a new element at index `n`.
149    *  @param  n     the index where a new element will be inserted.
150    *  @param  iter  the iterable object providing all elements to insert.
151    *  @throws Predef.IndexOutOfBoundsException if `n` is out of bounds.
152    */
153   def insertAll(n: Int, seq: Traversable[A]) {
154     try {
155       if (exported) copy()
156       var elems = seq.toList.reverse
157       len += elems.length
158       if (n == 0) {
159         while (!elems.isEmpty) {
160           val newElem = new :: (elems.head, start)
161           if (start.isEmpty) last0 = newElem
162           start = newElem
163           elems = elems.tail
164         }
165       } else {
166         var cursor = start
167         var i = 1
168         while (i < n) {
169           cursor = cursor.tail
170           i += 1
171         }
172         while (!elems.isEmpty) {
173           val newElem = new :: (elems.head, cursor.tail)
174           if (cursor.tail.isEmpty) last0 = newElem
175           cursor.asInstanceOf[::[A]].tl = newElem
```

```
176           elems = elems.tail
177         }
178       }
179     } catch {
180       case ex: Exception =>
181         throw new IndexOutOfBoundsException(n.toString())
182     }
183   }
184
185   /** Removes a given number of elements on a given index position. May take
186    *  time linear in the buffer size.
187    *  @param n         the index which refers to the first element to remove.
188    *  @param count     the number of elements to remove.
189    */
190   override def remove(n: Int, count: Int) {
191     if (exported) copy()
192     val n1 = n max 0
193     val count1 = count min (len - n1)
194     var old = start.head
195     if (n1 == 0) {
196       var c = count1
197       while (c > 0) {
198         start = start.tail
199         c -= 1
200       }
201     } else {
202       var cursor = start
203       var i = 1
204       while (i < n1) {
205         cursor = cursor.tail
206         i += 1
207       }
208       var c = count1
209       while (c > 0) {
210         if (last0 eq cursor.tail) last0 = cursor.asInstanceOf[::[A]]
211         cursor.asInstanceOf[::[A]].tl = cursor.tail.tail
212         c -= 1
213       }
214     }
215     len -= count1
216   }
217
218 // Implementation of abstract method in Builder
219
220   def result: List[A] = toList
221
222   /** Converts this buffer to a list. Takes constant time. The buffer is
223    *  copied lazily, the first time it is mutated.
224    */
225   override def toList: List[A] = {
226     exported = !start.isEmpty
227     start
228   }
229
230 // New methods in ListBuffer
231
232   /** Prepends the elements of this buffer to a given list
233    *
234    *  @param xs   the list to which elements are prepended
```

```
235     */
236   def prependToList(xs: List[A]): List[A] =
237     if (start.isEmpty) xs
238     else { last0.tl = xs; toList }
239
240 // Overrides of methods in Buffer
241
242   /** Removes the element on a given index position. May take time linear in
243    *  the buffer size.
244    *  @param  n  the index which refers to the element to delete.
245    *  @return n  the element that was formerly at position `n`.
246    *  @note      an element must exists at position `n`.
247    *  @throws Predef.IndexOutOfBoundsException if `n` is out of bounds.
248    */
249   def remove(n: Int): A = {
250     if (n < 0 || n >= len) throw new IndexOutOfBoundsException(n.toString())
251     if (exported) copy()
252     var old = start.head
253     if (n == 0) {
254       start = start.tail
255     } else {
256       var cursor = start
257       var i = 1
258       while (i < n) {
259         cursor = cursor.tail
260         i += 1
261       }
262       old = cursor.tail.head
263       if (last0 eq cursor.tail) last0 = cursor.asInstanceOf[::[A]]
264       cursor.asInstanceOf[::[A]].tl = cursor.tail.tail
265     }
266     len -= 1
267     old
268   }
269
270   /** Remove a single element from this buffer. May take time linear in the
271    *  buffer size.
272    *  @param x  the element to remove.
273    *  @return   this $coll.
274    */
275   override def -= (elem: A): this.type = {
276     if (exported) copy()
277     if (start.isEmpty) {}
278     else if (start.head == elem) {
279       start = start.tail
280       len -= 1
281     } else {
282       var cursor = start
283       while (!cursor.tail.isEmpty && cursor.tail.head != elem) {
284         cursor = cursor.tail
285       }
286       if (!cursor.tail.isEmpty) {
287         val z = cursor.asInstanceOf[::[A]]
288         if (z.tl == last0)
289           last0 = z
290         z.tl = cursor.tail.tail
291         len -= 1
292       }
293     }
```

```
294     this
295   }
296
297   override def iterator = new Iterator[A] {
298     var cursor: List[A] = null
299     def hasNext: Boolean = !start.isEmpty && (cursor ne last0)
300     def next(): A =
301       if (!hasNext) {
302         throw new NoSuchElementException("next␣on␣empty␣Iterator")
303       } else {
304         if (cursor eq null) cursor = start else cursor = cursor.tail
305         cursor.head
306       }
307   }
308
309   /** expose the underlying list but do not mark it as exported */
310   override def readOnly: List[A] = start
311
312   // Private methods
313
314   /** Copy contents of this buffer */
315   private def copy() {
316     var cursor = start
317     val limit = last0.tail
318     clear
319     while (cursor ne limit) {
320       this += cursor.head
321       cursor = cursor.tail
322     }
323   }
324
325   override def equals(that: Any): Boolean = that match {
326     case that: ListBuffer[_] => this.readOnly equals that.readOnly
327     case _                   => super.equals(that)
328   }
329
330   /** Returns a clone of this buffer.
331    *  @return a <code>ListBuffer</code> with the same elements.
332    */
333   override def clone(): ListBuffer[A] = (new ListBuffer[A]) ++= this
334
335   /** Defines the prefix of the string representation.
336    *  @return the string representation of this buffer.
337    */
338   override def stringPrefix: String = "ListBuffer"
339 }
340
341 /** $factoryInfo
342  *  @define Coll ListBuffer
343  *  @define coll list buffer
344  */
345 object ListBuffer extends SeqFactory[ListBuffer] {
346   implicit def canBuildFrom[A]: CanBuildFrom[Coll, A, ListBuffer[A]] =
347     new GenericCanBuildFrom[A]
348   def newBuilder[A]: Builder[A, ListBuffer[A]] =
349     new GrowingBuilder(new ListBuffer[A])
350 }
```