29. Bundeswettbewerb Informatik 2010/2011
2. Runde

Tim Taubner, Verwaltungsnummer 29.108.01

10. April 2011

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

Inhaltsverzeichnis

A. Allgemeines

1. Persönliche Anmerkungen

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

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

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

2. Dateistruktur der CD

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

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

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

3. Ausführvoraussetzungen

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

Java Runtime: Mindestens Version 1.5 (Java 5), empfohlen: $\geq$ 1.6

Prozessor: Mindestens 1 GHz, empfohlen: $\geq$ 1.6 GHz

Arbeitsspeicher: Mindestens 512 MB, empfohlen: $\geq$ 1 GB

Grafikkarte: beliebig

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

[1] siehe Einsendung zur 1. Runde, Einsendungsnr. 108, besonders Aufgabe 1
[2] Hier zeigt sich eine Schwäche der deutschen Sprache: Sie ist *verbose*. Ich versuche hier *concise* aus dem Englischem zu übersetzen
[3] Vergleichen Sie dies mit einem Eisberg, er zieht beim schmilzen die ganze Wärme aus der Umgebung.
[4] Meine eigene Perle der Informatik ;-]

4. Starten der Programme

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

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

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

java -jar Aufgabe1/target/Kisten.jar

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

java -jar Aufgabe1/target/Kisten.jar

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

java -jar Aufgabe1/target/Kisten.jar

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

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

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

1. Lösungsidee

1.1. Allgemein

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

1.2. Bruteforce

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

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

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

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

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

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

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

1.3. Bruteforce nach Aufteilung

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

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

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

.

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

1.4. Online Packer

Ansatz Ein etwas anderer Ansatz ist, Kistensätze inkrementiell zu erzeugen. Daher, es wird ein Algorithmus erfordert, welcher zu einem - mehr oder weniger gut - gepacktem Kistensatz und einer zu packenden Kiste ein neuen Kistensatz liefert, der, möglichst dicht gepackt, diese enthält. Eine weitere Beschränkung, die ich an den Algorithmus stelle, ist, dass er in höchstens $O(n)$ Zeit diesen neuen Kistensatz liefert. Hat man nun solch einen Algorithmus, lassen sich alle Kisten zusammen in $O(n \cdot n) = O(n^2)$ Zeit packen bei inkrementieller Kistensatzerzeugung.

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

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

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

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

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

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

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

2. Implementierung

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

[12] Siehe auch: 3.1

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

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

1. Das Ersetzen eines [Teil]baumes sollte möglichst billig sein. Da das Ersetzen einer Kiste als Löschen und anschließendes Hinzufügen dieser in den Baum implementiert ist, müssen sowohl die Lösch- als auch Hinzufügefunktionen 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 billig ist.

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

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

3. Programmabläufe

3.1. Algorithm-Contest (Packdichte)

3.2. Algorithm-Contest (Laufzeit)

4. Programmtext

5. Programmnutzung

C. Zweite bearbeitete Aufgabe: (2) Containerklamüsel

1. Lösungsidee

1.1. Vorüberlegungen

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

1. $\pi^k(i) = i$,
2. die Elemente $i, \pi(i), \pi^2(i), \ldots, \pi^{k-1}(i)$ sind paarweise verschieden,
3. jedes Element, das verschieden von $i, \pi(i), \pi^2(i), \ldots, \pi^{k-1}(i)(= i)$ 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

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

[...]

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

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

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

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

1.2. Datenstruktur

Permutationen können in einer indexierten Liste jeder Art (beispielsweise einem Array) gespeichert werden. Da in der Informatik jedoch indexierte Listen (insbesondere Arrays) meist Indizes aus $[0, n[$ besitzen muss dies beim Zugriff beachtet werden. Um also die Zahl p zu finden, auf die i durch $perm$ abgebildet wird, gilt $p = perm(i-1)$ jedoch nicht $p = perm(i)$. Es wird außerdem noch eine einfache Datenstruktur benötigt, um das Gleis mit Containerstellplätzen und Waggons abzubilden. Diese wird in 2.3 noch genauer erläutert.

1.3. Ergebnisoptimaler Algorithmus

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

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

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

```scala
def cyclesOf(perm: Seq[Int], ready: Set[Int]): List[List[Int]] =
  (1 to perm.length) find (i => !ready.contains(i)) match {
      case Some(start) =>
        val newCycle = cycle(perm, start)
        val newReady = ready ++ newCycle // O(n)
        newCycle :: cyclesOf(perm, newReady)
      case None =>
        Nil
    }
```

// TODO: REWRITE!
Anhand der berechneten Zyklen wird im nächsten Schritt die Instruktionskette errechnet. Folgend ist der Scalacode abgebildet, welcher die Instruktionen berechnet.
Bemerkung: Um den Code gut in der Dokumentenzeilenbreite darstellen zu können, wurden die “type aliases” Cycle für List[Int], Cycles für List[Cycle] und, speziell für die Hilfsfunktion step Step für (ListBuffer[Instruction], Cycles, Int).

```
def computeFromCycles(cycles: Cycles): Seq[Instruction] = {
  // Füge erstes TakeCon hinzu, damit bereits ein Container auf dem Kran ist,
  // Lösche letzes Element (PutWag) mit init
  TakeCon :: computeCycle(cycles.head, cycles.tail).init._1.toList
}

def computeCycle(cycle: Cycle, other: Cycles): (ListBuffer[Instruction], Cycles) = {
    val max = cycle.max

    type Step = (ListBuffer[Instruction], Cycles, Int)

    def step(instrs: ListBuffer[Instruction], cyclesLeft: Cycles,
             prev: Int, cur: Int): Step =
      cyclesLeft.headOption match {
        // Gibt es ein nächsten Zyklus und beginnt dieser direkt nach diesem?
        case Some(nextCycle @ (next :: _)) if prev == max && max+1 == next => // (1)
          // Wenn ja, "konkateniere" diese.
          val (cycleInstrs, newCyclesLeft) =
            computeCycle(cyclesLeft.head, cyclesLeft.tail)
          val extraInstrs = instrs ++=
            ListBuffer(PutCon, MoveRight, TakeCon) ++=
            cycleInstrs ++= ListBuffer(MoveLeft, TakeCon)
          step(extraInstrs, newCyclesLeft, prev, cur)
        // Gibt es ein nächsten Zyklus und beginnt dieser vor dem nächsten Element?
        case Some(nextCycle @ (next :: _)) if next < cur => // (2)
          // Wenn ja, dann arbeite erst nextCycle ab.
          val (transInstrs, newCyclesLeft) = computeCycle(nextCycle, cyclesLeft.tail)
          val newInstrs = instrs ++=
            ListBuffer(Move(prev -> next), Rotate, TakeCon, Rotate, PutCon, Rotate) ++=
            transInstrs
          step(newInstrs, newCyclesLeft, next, cur)
        case _ => // (3)
          // Andernfalls, fahre einfach mit der Abarbeitung fort.
          val newInstrs = instrs ++=
            ListBuffer(Move(prev -> cur), Rotate, PutWag, TakeCon)
          (newInstrs, cyclesLeft, cur)
      }

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

Dieser Algorithmus ist deutlich komplexer als die vorherigen. Deswegen wurden Kommentare hinzugefügt. Im Allgemeinen soll computeCycle für einen Zyklus ϕ_x und die restlichen Zyklen $\phi_{x+1}, \ldots, \phi_o$ die Instruktionen für einen Weg liefern, so dass alle Elemente der gegebenen Zyklen an die richtige Position gebracht werden und der Kran wieder an *TODO!*

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

Zunächst wird die Korrektheit der Hilfsfunktion cycle gezeigt. Das heißt, wir vergewissern und, dass cycle zu einer gegebenen Permutation π immer den Zyklus ϕ findet, der an dem Startindex i beginnt. Es ist also ein Zyklus ϕ der folgenden Form gesucht.

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

Für alle x die im Zyklus ϕ enthalten sind gilt $\phi(x) = \pi(x)$. Weiter sind genau die Elemente $i, \phi(i), \phi^2(i), \dots, \phi^{k-i}(i)$ enthalten, also gilt

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

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

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

Im Folgenden können wir uns also der Korrektheit von cycle sicher sein. Nun soll die Korrektheit von cyclesOf gezeigt werden. Wir wollen also beweisen, dass cyclesOf zu einer gegebenen Permutation π und einer leeren Menge von "fertigen" Elementen eine Liste von disjunkten Zyklen $\phi_1, \phi_2, \dots, \phi_o$ zurückgibt, wobei o die Anzahl disjunkter Zyklen ist und $x < y \Leftrightarrow \ min(\phi_x) < min(\phi_y)$ für alle $x, y = 0 \dots o$. Es sollen also nach Startwert sortierte Zyklen zurückgeliefert werden. Es wird im folgenden wieder Induktion verwendet. Im Induktionsanfang soll also gezeigt werden, dass cyclesOf für $ready = \phi_1 \cup \phi_2 \cup \dots \cup \phi_o$ alle verbleibende Zyklen - nämlich gar keine - findet. Da $\|ready\| = \|\phi_1 \cup \dots \cup \phi_o\|$, bricht cyclesOf ab mit der leeren Liste. Dies ist korrekt, denn es sind bereits alle Zyklen gefunden. Nun gelte, dass cyclesOf für ein $x \in \{0, \dots, o\}$ und $ready = \phi_1 \cup \phi_2 \cup \dots \cup \phi_{x-1} \cup \phi_x$ die Zyklen $\phi_{x+1}, \dots, \phi_o$ findet. Wir zeigen, dass dies auch für $x \to x - 1$ gilt. Zunächst wird der Wert $s \in \{1, \dots, n\}$ ($s =$start, n ist die Länge von π) mit $s \notin ready$ gesucht. Nun wird der neue Zyklus ϕ_x berechnet. Dieser ist sicher disjunkt von den zuvor berechneten Zyklen, da er bei $s \notin ready$ beginnt. Anschließend wird cyclesOf rekursiv aufgerufen, mit $ready = \phi_1 \cup \dots \cup \phi_{x-1} \cup \phi_x$. Dieser Aufruf liefert nach Induktionsannahme die Zyklen $\phi_{x+1}, \dots, \phi_o$. Also werden insgesamt die Zyklen $\phi_x, \phi_{x+1}, \dots, \phi_o$ ausgegeben. Das auch die Sortierung richtig ist, sieht man anhand der Tatsache, dass immer der kleinstmögliche Startwert gesucht wird. Also ist auch dieser Algorithmus korrekt, bei Aufruf von cyclesOf mit $ready =$ werde nämlich die Zyklen $\phi_1, \dots, \phi_o$ zurückgegeben.

Anschließend zeigen wir die Optimalheit vom eigentlichem Algorithmus, die Berechnung der Instruktionen.

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

1.4. Optimaler Algorithmus

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

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

```
@tailrec private def cyclesOf(ready: ListBuffer[Cycle], perm: Seq[Int],
                               handled: Array[Boolean], start: Int = 1): Cycles = //
    if(start > perm.length) ready.toList
    else {
      val aCycle = cycle(perm, start) // O(n_c)
      for (i ← aCycle) { handled(i-1) = true } // O(n_c); Side effects are not harmful
      (start to perm.length) find (i ⇒ !(handled(i-1))) match { // O(i_c) //\sum{i_c} =
        case Some(next) ⇒
          cyclesOf(ready += aCycle, perm, handled, next)
        case None ⇒
          (ready += aCycle).toList // O(1)
      }
    }
```

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

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

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

2. Implementierung

Die Implementierung gliedert sich folgendermaßen.

cycler Algorithmen zur Berechnung der Zyklen (Sowohl langsamerer, als auch schnellerer)

Instructor Algorithmus zur Berechnung der Instruktionen aus den Zyklen

Gleis Datenstruktur zur Verwaltung der Containerstellplätze und Waggons

Maschine Klasse zur Simulation einer Maschine

ListBuffer Modifizierte Variante des standardmäßigem Scala ListBuffer

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

2.1. Cycler - Berechnung der Zyklen

Da beide Algorithmen zur Zyklenfindung implementiert werden sollen, wurde zunächst das **trait** Cycler implementiert, welches die einzige Methode des Moduls cyclesOf(Seq[Int]): List [List [Int]] definiert. Diese soll zu einer gegebenen Permutation eine Liste von nach Startelementen sortierte Zyklen zurückgeben.
Die Implementierung des SlowCycler erfolgte wie in 1.3, die des FastCycler nach 1.4.

2.2. Instructor - Berechnung der Instruktionen

Anschließend wurden im Modul Instructor Funktionen zur Berechnung der Instruktionen erstellt. Diese gliedern sich in die von "außen" zu benutzenden Funktionen sowie die "innen" benötigten Hilfsfunktionen. Von außen sind compute(Seq[Int], Cycler): Seq[Instruction] und computeFromCycles(List[List[Int]]): Seq[Instruction] zu benutzen. Die letztere berechnet die Liste der Instruktionen aus (meist vorher berechneten) Zyklen, während die erstere die Benutzung dadurch vereinfacht, nur die Permutation angeben zu müssen (die Zyklen werden dann automatisch berechnet). Die "inneren" Hilfsfunktionen sind folgende.
computeCycle(List[Int], List [List [Int]]): (ListBuffer [Instruction], List [List [Int]]) gibt zu einem zu bearbeitenden Startzyklus und restlichen Zyklen eine Liste von Instruktionen und eine Liste von unbearbeiteten Zyklen zurück.

2.3. Gleis - Speichern des Zustand

Die Datenstruktur zum Speichern des aktuellen Status der Container, Containerstellplätzen und Waggons wird in der Klasse Gleis implementiert. Ein Gleis verwaltet zwei Arrays der Länge n. Das erste Array con speichert die jeweilige Containernummer auf dem zugehörigen Containerstellplatz. Das andere Array wag speichert die jeweilige Nummer des Container auf einem Waggon. Zu Beginn wird das Array con mit der Permutation initialisiert.
Ein Gleis stellt die Methoden takeCon(Int): Int, takeWag(Int): Int, putCon((Int, Int)): Int und putWag((Int, Int)): Int. Außerdem wurde die toString: String Methode überschrieben, um eine formatierte Ausgabe zu erhalten. Die oben genannten Methoden sind zur Manipulation der Containernummern zu den jewiligen Containerstellplätzen, bzw. Waggons da. Genauere Verwendung wird bei späterer Referenz genauer beschrieben.

2.4. Maschine - Interpretieren der Instruktionen

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

2.5. ListBuffer - Erweiterung einer Standardklasse

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

2.6. Utils - Helfende Methoden

Weitere Methoden, die nützlich im Rahmen der Nutzung des Programmes sind, jedoch nicht direkt zur Implementierung der Aufgabelösung dienen, wurden in das Modul Utils ausgelagert. Besondere Bedeutung hat die Funktion randPerm, die zu einer gegebenen Permutationslänge eine zufällige Permutation berechnet. Außerdem wurden auch Methoden zum Speichern der Instruktionsketten und Permutationen implementiert.

3. Programmabläufe

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

Beispiel aus der Aufgabenstellung Folgend ist der Ablauf der sich bei Eingabe des Beispiels aus der Aufgabestellung ergibt dargestellt.
Zunächst wird die Permutation erzeugt und in perm gespeichert.

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

Anschließend werden die Instruktionen erzeugt und in instrs gespeichert.

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

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

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

Zuletzt soll die Maschine die Instruktionen interpretieren.

```
scala> maschine interpret instrs
1 2 3 4;(m=8)
4 3 2 1;(l=8)
⟶        (1)
  ⟶      (1)
  ⟵      (1)
  ⟶⟶     (2)
⟵⟵⟵      (3)
res0: de.voodle.tim.bwinf.container.Gleis =
Container: _ _ _ _
Waggons:   1 2 3 4
```

Bemerkenswert ist hier, dass der erstellte Algorithmus in diesem Fall exakt den gleichen Weg liefert wie im Beispiel der Aufgabenstellung angegeben. Es gibt noch verschiedene andere Wege. Beispielsweise kann das Prüfen auf überlappende Zyklen erst beim Zurückfahren erfolgen. Andere Möglichkeiten für einen optimalen Weg wären folgend dargestellte Abläufen. Es gibt also insgesamt vier verschieden Fahrpläne, die für das Beispiel einen optimalen Weg ergeben.

```
1 2 3 4;(m=8)    1 2 3 4;(m=8)    1 2 3 4;(m=8)
4 3 2 1;(l=8)    4 3 2 1;(l=8)    4 3 2 1;(l=8)
⟶⟶⟶      (3)     ⟶⟶⟶      (3)     ⟶⟶       (2)
    ⟵    (1)       ⟵⟵     (2)       ⟵      (1)
  ⟵      (1)       ⟶      (1)       ⟶      (1)
  ⟶      (1)       ⟵      (1)         ⟶    (1)
⟵⟵       (2)     ⟵        (1)     ⟵⟵⟵      (3)
```

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

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

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

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

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

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

scala> maschine interpret instrs
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20;(m=138)
 20  11   2   8   1  16  10  17  19  14   5  12   9   3  13  15  18   4   7   6;(l=138)
 ------------>                                                                 (3)
             ---------------->                                                 (4)
                             ---------------->                                 (4)
                                             <                                  (0)
                                             -------------------->             (5)
                                                                 ---->         (1)
             <--------------------------------------------------------         (14)
             ---------------------------------------------------------------->  (16)
                     <--------------------------------------------------------  (14)
                     ---------------------------------------->                 (10)
                                                         <---                  (1)
                                                 <-------                      (2)
                                 <---------------                              (4)
                                 ---------------------------------------->     (10)
                         <------------------------------------------------     (12)
                         ------------>                                         (3)
                                     ---------------->                         (4)
         <--------------------------------------------                         (11)
     <---                                                                      (1)
     ------------------------------------>                                     (9)
                 <------------------------                                     (6)
 <----------------                                                             (4)
res0: de.voodle.tim.bwinf.container.Gleis =
Container: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
Waggons:   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

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

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

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

Interessant ist hier die Beobachtung, dass es nur 18 Zyklen gibt, bei einer Permutationslänge von 10^7. Insgesamt wurden 110879 Millisekunden, also 110 Sekunden bzw. knapp 2 Minuten benötigt, um die Instruktionen zu berechen. Dies ist ein Indiz auf oben bewiesene gute Laufzeitkomplexität. Nach der Berechnung der Instruktionen wurden diese testweise interpretiert. Hierfür wurden knapp 11 Sekunden benötigt. Zum Schluß wurde außerdem verifiziert, dass jeder Container auf der richtigen Position ist.

4. Programmnutzung

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

```
scala>import de.voodle.tim.bwinf.container._
import de.voodle.tim.bwinf.container._
```

4.1. Permutationen erzeugen

Permutationen erzeugen Sie entweder durch direkte Eingabe oder Sie lassen eine randomisierte Permutation für eine gegebene Länge erzeugen. Um eine Permutation direkt einzugeben können Sie einfach die Hilfsfunktionen der Scalabibliothek benutzen. Speichern Sie einfach das Bild der Permutation in einer Seq. Die Permutation aus der Aufgabenstellung geben Sie beispielsweise wie folgt ein.

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

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

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

4.2. Erzeugen der Instruktionen

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

```
scala> val instrs = Instructor compute perm
instrs: Seq[de.voodle.tim.bwinf.container.Instruction] = List(TakeCon, MoveRight(1), .
```

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

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

```
scala> val instrs = Instructor computeFromCycles cycles
instrs: Seq[de.voodle.tim.bwinf.container.Instruction] = List(TakeCon, MoveRight(1), .
```

4.3. Simulation der Maschine

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

```
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
```

4.4. Zeitmessung

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

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

5. Programmtext

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

5.1. Cycler

```scala
package de.voodle.tim.bwinf.container
import annotation.tailrec
import scala.collection.mutable.tim.ListBuffer // <-- custom ListBuffer

object Cycler {
  type Cycle = List[Int]
  type Cycles = List[List[Int]]
}
import Cycler._
trait Cycler extends Function1[Seq[Int], List[List[Int]]] {
  def apply(perm: Seq[Int]) = cyclesOf(perm)
  def cyclesOf(perm: Seq[Int]): Cycles
}
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], ready: Set[Int]): Cycles =
    (1 to perm.length) find (i => !ready.contains(i)) match {
      case Some(start) =>
        val newCycle = cycle(perm, start)
        val newReady = ready ++ newCycle // O(n)
        newCycle :: cyclesOf(perm, newReady)
      case None =>
        Nil
    }
}
object FastCycler extends Cycler {
  /** Return the list of disjunct cycles sorted ascending by cycle.head */
  def cyclesOf(perm: Seq[Int]): Cycles =
    cyclesOf(new ListBuffer[Cycle], perm, new Array[Boolean](perm.length))

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

5.2. 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.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): (ListBuffer[Instruction], Cycles) = {
    val max = cycle.max // O(n_c)

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

5.3. Gleis

```
package de.voodle.tim.bwinf.container

class Gleis(initCon: Seq[Int]) {
  val length = initCon.length
  private val con = Seq(initCon: _*).toArray
  private val wag = new Array[Int](length)

  private def arrTake(arr: Array[Int])(i: Int): Int = {
    val res = arr(i-1)
    arr(i-1) = 0
    res
  }
  private def arrPut(arr: Array[Int])(map: (Int, Int)) = map match {
    case (i, what) =>
      require(arr(i-1) == 0, "arr(i-1) at " + i + " must be 0, but is " + arr(i-1))
      arr(i-1) = what
  }

  def takeWag(i: Int) = arrTake(wag)(i)
  def takeCon(i: Int) = arrTake(con)(i)
  def putWag(map: (Int, Int)) = arrPut(wag)(map)
  def putCon(map: (Int, Int)) = arrPut(con)(map)

  private def arrString(arr: Array[Int]) = // Only print first 100
    arr take 100 map (i => if(i == 0) "_" else i.toString) mkString " "
  override def toString =
     "Container: " + arrString(con) + "\n" +
     "Waggons:   " + arrString(wag)

  // Immutable Vector copies!
  def container = Vector(con: _*)
  def waggons = Vector(wag: _*)
}
```

5.4. Maschine

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

class Maschine(protected val gleis: Gleis,
               private val print: Boolean = false) {
  import Maschine._
  private val length = gleis.length
  private val numLength = digits(length)
  private val space = " " * (numLength+1)
  private val arrow = "-" * (numLength+1)

  private def minLength =
    gleis.container.zipWithIndex.map { case (v,i) => ((i+1)-v).abs } sum

  def log(str: =>Any) = if(print) println(str) else ()

  def logInts(ints: =>Seq[Int]): String =
    (for(i <- ints) yield {
        val diff = numLength - digits(i)
        " " * diff + i
      }) mkString (" ")

  def interpret(instrs: Seq[Instruction]): Gleis = {
    log(logInts(1 to length) + ";(m=" + minLength + ")")
    log(logInts(gleis.container) + ";(l=" + instrs.map(_.len).sum + ")")
    interpret(instrs.toList,0,0,1)
  }
  // Attach point for further actions (for subclasses)
  protected def act(instrs: List[Instruction]) {}

  @tailrec
  private def interpret(instrs: List[Instruction], con: Int, wag: Int, idx: Int): Gleis = {
    act(instrs)
    instrs match { // Recursivly check
      case Rotate :: xs =>
        interpret(xs,wag,con,idx)
      case TakeCon :: xs =>
        interpret(xs, gleis takeCon idx, wag, idx)
      case TakeWag :: xs =>
        interpret(xs, 0, gleis takeWag idx, idx)
      case PutCon :: xs =>
        gleis putCon (idx -> con)
        interpret(xs, 0, wag, idx)
      case PutWag :: xs =>
        gleis putWag (idx -> wag)
        interpret(xs, con, 0, idx)
      case MoveRight(len) :: xs =>
        log(space * (idx-1) + arrow * len + ">" +
            space * (length-len-idx) + " (" + len + ")")
        interpret(xs, con, wag, idx+len)
      case MoveLeft(len) :: xs =>
        log(space * (idx-1-len) + "<" + arrow * len +
            space * (length-idx) + " (" + len + ")")
        interpret(xs, con, wag, idx-len)
      case Nil => gleis // Do Nothing
    }
  }
  override def toString = gleis.toString
}
object Maschine {
  private def digits(num: Int) = (math.log10(num) + 1).floor.toInt
}
```

5.5. Instructions

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

sealed trait Instruction {
  def len: Int = 0
  def short: String = "" + toString.head
}
case object PutWag extends Instruction
case object PutCon extends Instruction
case object Rotate extends Instruction
case object TakeWag extends Instruction
case object TakeCon extends Instruction
sealed trait Move extends Instruction {
  override def short = (toString filter (_.isUpper)) + "(" + len + ")"
}
object Move {
  def apply(len: Int): Move =
    if(len > 0) MoveRight(len)
    else        MoveLeft(-len)
  def apply(fromTo: (Int, Int)): Move = fromTo match {
    case (from,to) => Move(to - from)
  }
}
case class MoveLeft(override val len: Int) extends Move
object MoveLeft extends MoveLeft(1)
case class MoveRight(override val len: Int) extends Move
object MoveRight extends MoveRight(1)
```

5.6. Utils

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

object Utils {
  import scala.util.Random
  import scala.collection.mutable.IndexedSeq
  def randPerm(n: Int) = {
    // Make sure we don't convert it to an WrappedArray to often.
    val a: IndexedSeq[Int] = new Array[Int](n)
    // Init array // O(n)
    for (idx <- 0 until n) a(idx) = idx + 1
    // randomize array // O(n)
    for (i <- n to 2 by -1) {
      val di = Random.nextInt(i)
      val swap = a(di)
      a(di) = a(i-1)
      a(i-1) = swap
    }
    a // return array
  }

  def demonstrate(n: Int) = {
    val startTime = System.currentTimeMillis
    val perm = randPerm(n)
    val cycles = FastCycler cyclesOf perm
    println("Time used for computing Cycles: " + (System.currentTimeMillis - startTime))
    println("Number of cycles: " + cycles.length)
    val instrs = Instructor computeFromCycles cycles
    val endTime = System.currentTimeMillis
    println("Time used: " + (endTime - startTime))
    val gleis = new Gleis(perm)
    val maschine = new Maschine(gleis)
    maschine interpret instrs
    println("Time used interpreting: " + (System.currentTimeMillis - endTime))
    println("Verifying results...")
    gleis.waggons.zipWithIndex forall (xy => xy._1 == xy._2 + 1)
  }
}
```

5.7. ListBuffer

```scala
package scala.collection.mutable.tim
import scala.collection.{mutable,generic,immutable}
import mutable._
import generic._
import immutable.{List, Nil, ::}

/** A `Buffer` implementation back up by a list. It provides constant time
 *  prepend and append. Most other operations are linear.
 *
 *  @author  Tim Taubner
 *  @author  Matthias Zenger
 *  @author  Martin Odersky
 *  @version 2.8.tim
 *  @since   1
 *
 *  @tparam A    the type of this list buffer's elements.
 *
 *  @define Coll ListBuffer
 *  @define coll list buffer
 *  @define thatinfo the class of the returned collection. In the standard library configuration,
 *    `That` is always `ListBuffer[B]` because an implicit of type `CanBuildFrom[ListBuffer, B, ListB
 *    is defined in object `ListBuffer`.
 *  @define $bfinfo an implicit value of class `CanBuildFrom` which determines the
 *    result class `That` from the current representation type `Repr`
 *    and the new element type `B`. This is usually the `canBuildFrom` value
 *    defined in object `ListBuffer`.
 *  @define orderDependent
 *  @define orderDependentFold
 *  @define mayNotTerminateInf
 *  @define willNotTerminateInf
 */
@serializable @SerialVersionUID(3419063961353022661L)
final class ListBuffer[A]
      extends Buffer[A]
         with GenericTraversableTemplate[A, ListBuffer]
         with BufferLike[A, ListBuffer[A]]
         with Builder[A, List[A]]
         with SeqForwarder[A]
{
  override def companion: GenericCompanion[ListBuffer] = ListBuffer

  import scala.collection.Traversable

  private var start: List[A] = Nil
  private var last0: ::[A] = _
  private var exported: Boolean = false
  private var len = 0

  protected def underlying: immutable.Seq[A] = start

  /** The current length of the buffer.
   *
   *  This operation takes constant time.
   */
  override def length = len

  // Implementations of abstract methods in Buffer

  override def apply(n: Int): A =
    if (n < 0 || n >= len) throw new IndexOutOfBoundsException(n.toString())
    else super.apply(n)

  /** Replaces element at index `n` with the new element
   *  `newelem`. Takes time linear in the buffer size. (except the
   *  first element, which is updated in constant time).
   *
   *  @param n  the index of the element to replace.
```

```
   *  @param x  the new element.
   *  @throws Predef.IndexOutOfBoundsException if `n` is out of bounds.
   */
  def update(n: Int, x: A) {
    try {
      if (exported) copy()
      if (n == 0) {
        val newElem = new :: (x, start.tail);
        if (last0 eq start) {
          last0 = newElem
        }
        start = newElem
      } else {
        var cursor = start
        var i = 1
        while (i < n) {
          cursor = cursor.tail
          i += 1
        }
        val newElem = new :: (x, cursor.tail.tail)
        if (last0 eq cursor.tail) {
          last0 = newElem
        }
        cursor.asInstanceOf[::[A]].tl = newElem
      }
    } catch {
      case ex: Exception => throw new IndexOutOfBoundsException(n.toString())
    }
  }

  // THIS PART IS NEW (by tim8dev):

  /** Appends a single element to this buffer. This operation takes constant time.
   *
   *  @param x  the element to append.
   *  @return   this $coll.
   */
  def += (x: A): this.type = {
    val newLast = new :: (x,Nil)
    append(newLast, newLast, 1)
  }

  override def ++=(xs: TraversableOnce[A]): this.type = xs match {
    case some : ::[A] =>
      append(some, some.last.asInstanceOf[::[A]], some.length)
    case buff : ListBuffer[A] =>
      buff.start match {
        case some : ::[A] =>
          if(buff.exported)
            buff.copy()
          buff.exported = true
          append(some, buff.last0, buff.len)
        case Nil =>
          this
      }
    case xs =>
      super.++=(xs)
  }

  private def append(x: ::[A], last: ::[A], length: Int): this.type = {
    if(exported) copy()
    if(start.isEmpty) {
      last0 = last
      start = x
    } else {
      val last1 = last0
      last1.tl = x
      last0 = last
    }
```

```
    len += length
    this
  }

  // END OF NEW PART (by tim8dev).

  /** Clears the buffer contents.
   */
  def clear() {
    start = Nil
    exported = false
    len = 0
  }

  /** Prepends a single element to this buffer. This operation takes constant
   *  time.
   *
   *  @param x  the element to prepend.
   *  @return   this $coll.
   */
  def +=: (x: A): this.type = {
    if (exported) copy()
    val newElem = new :: (x, start)
    if (start.isEmpty) last0 = newElem
    start = newElem
    len += 1
    this
  }

  /** Inserts new elements at the index `n`. Opposed to method
   *  `update`, this method will not replace an element with a new
   *  one. Instead, it will insert a new element at index `n`.
   *
   *  @param  n     the index where a new element will be inserted.
   *  @param  iter  the iterable object providing all elements to insert.
   *  @throws Predef.IndexOutOfBoundsException if `n` is out of bounds.
   */
  def insertAll(n: Int, seq: Traversable[A]) {
    try {
      if (exported) copy()
      var elems = seq.toList.reverse
      len += elems.length
      if (n == 0) {
        while (!elems.isEmpty) {
          val newElem = new :: (elems.head, start)
          if (start.isEmpty) last0 = newElem
          start = newElem
          elems = elems.tail
        }
      } else {
        var cursor = start
        var i = 1
        while (i < n) {
          cursor = cursor.tail
          i += 1
        }
        while (!elems.isEmpty) {
          val newElem = new :: (elems.head, cursor.tail)
          if (cursor.tail.isEmpty) last0 = newElem
          cursor.asInstanceOf[::[A]].tl = newElem
          elems = elems.tail
        }
      }
    } catch {
      case ex: Exception =>
        throw new IndexOutOfBoundsException(n.toString())
    }
  }

```

```
  /** Removes a given number of elements on a given index position. May take
   *  time linear in the buffer size.
   *
   *  @param n          the index which refers to the first element to remove.
   *  @param count      the number of elements to remove.
   */
  override def remove(n: Int, count: Int) {
    if (exported) copy()
    val n1 = n max 0
    val count1 = count min (len - n1)
    var old = start.head
    if (n1 == 0) {
      var c = count1
      while (c > 0) {
        start = start.tail
        c -= 1
      }
    } else {
      var cursor = start
      var i = 1
      while (i < n1) {
        cursor = cursor.tail
        i += 1
      }
      var c = count1
      while (c > 0) {
        if (last0 eq cursor.tail) last0 = cursor.asInstanceOf[::[A]]
        cursor.asInstanceOf[::[A]].tl = cursor.tail.tail
        c -= 1
      }
    }
    len -= count1
  }

// Implementation of abstract method in Builder

  def result: List[A] = toList

  /** Converts this buffer to a list. Takes constant time. The buffer is
   *  copied lazily, the first time it is mutated.
   */
  override def toList: List[A] = {
    exported = !start.isEmpty
    start
  }

// New methods in ListBuffer

  /** Prepends the elements of this buffer to a given list
   *
   *  @param xs   the list to which elements are prepended
   */
  def prependToList(xs: List[A]): List[A] =
    if (start.isEmpty) xs
    else { last0.tl = xs; toList }

// Overrides of methods in Buffer

  /** Removes the element on a given index position. May take time linear in
   *  the buffer size.
   *
   *  @param  n  the index which refers to the element to delete.
   *  @return n  the element that was formerly at position `n`.
   *  @note      an element must exists at position `n`.
   *  @throws Predef.IndexOutOfBoundsException if `n` is out of bounds.
   */
  def remove(n: Int): A = {
    if (n < 0 || n >= len) throw new IndexOutOfBoundsException(n.toString())
    if (exported) copy()
```

```
    var old = start.head
    if (n == 0) {
      start = start.tail
    } else {
      var cursor = start
      var i = 1
      while (i < n) {
        cursor = cursor.tail
        i += 1
      }
      old = cursor.tail.head
      if (last0 eq cursor.tail) last0 = cursor.asInstanceOf[::[A]]
      cursor.asInstanceOf[::[A]].tl = cursor.tail.tail
    }
    len -= 1
    old
  }

  /** Remove a single element from this buffer. May take time linear in the
   *  buffer size.
   *
   *  @param x  the element to remove.
   *  @return   this $coll.
   */
  override def -= (elem: A): this.type = {
    if (exported) copy()
    if (start.isEmpty) {}
    else if (start.head == elem) {
      start = start.tail
      len -= 1
    } else {
      var cursor = start
      while (!cursor.tail.isEmpty && cursor.tail.head != elem) {
        cursor = cursor.tail
      }
      if (!cursor.tail.isEmpty) {
        val z = cursor.asInstanceOf[::[A]]
        if (z.tl == last0)
          last0 = z
        z.tl = cursor.tail.tail
        len -= 1
      }
    }
    this
  }

  override def iterator = new Iterator[A] {
    var cursor: List[A] = null
    def hasNext: Boolean = !start.isEmpty && (cursor ne last0)
    def next(): A =
      if (!hasNext) {
       throw new NoSuchElementException("next on empty Iterator")
      } else {
        if (cursor eq null) cursor = start else cursor = cursor.tail
        cursor.head
      }
  }

  /** expose the underlying list but do not mark it as exported */
  override def readOnly: List[A] = start

  // Private methods

  /** Copy contents of this buffer */
  private def copy() {
    var cursor = start
    val limit = last0.tail
    clear
    while (cursor ne limit) {
```

```
      this += cursor.head
      cursor = cursor.tail
    }
  }

  override def equals(that: Any): Boolean = that match {
    case that: ListBuffer[_] => this.readOnly equals that.readOnly
    case _                   => super.equals(that)
  }

  /** Returns a clone of this buffer.
   *
   *  @return a <code>ListBuffer</code> with the same elements.
   */
  override def clone(): ListBuffer[A] = (new ListBuffer[A]) ++= this

  /** Defines the prefix of the string representation.
   *
   *  @return the string representation of this buffer.
   */
  override def stringPrefix: String = "ListBuffer"
}

/** $factoryInfo
 *  @define Coll ListBuffer
 *  @define coll list buffer
 */
object ListBuffer extends SeqFactory[ListBuffer] {
  implicit def canBuildFrom[A]: CanBuildFrom[Coll, A, ListBuffer[A]] = new GenericCanBuildFrom[A]
  def newBuilder[A]: Builder[A, ListBuffer[A]] = new GrowingBuilder(new ListBuffer[A])
}
```