Algorithmen und Datenstrukturen
Übungsblatt 04

Prof. Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

SoSe 2022	v2.0.2
Themen:	Sortieralgorithmen
Relevante Foliensätze:	Foliensätze/Video zum Sortieren
Abgabe der Hausübung:	03.06.2022 bis 23:50 Uhr

Hausübung 04 — Gesamt: 32 Punkte

Hybride Sortieralgorithmen

Verbindliche Anforderungen für alle Hausübungen:

Das Dokumentieren und Kommentieren Ihres Quelltextes ist nicht verbindlich, wird zum besseren Verständnis Ihrer Lösung jedoch empfohlen. Alle zur Bewertung dieser Hausübung relevanten Deklarationen von Klassen, Methoden (hierzu zählen auch Konstruktoren) und Attributen sind bereits in der Quelltext-Vorlage enthalten und dürfen nicht modifiziert oder entfernt werden. Ihnen steht aber frei, Hilfskonstrukte in Form von weiteren Klassen, Methoden und Attributen zu erstellen, sofern dies nicht explizit auf dem Übungsblatt verboten wurde und Ihre Hilfskonstrukte nicht gegen verbindliche Anforderungen verstoßen. Datenstrukturen und Hilfsmethoden aus der Java-Standardbibliothek sowie Arrays sind nicht erlaubt, sofern dies nicht explizit auf dem Übungsblatt gefordert oder erlaubt wurde. Ihre Methoden müssen auch dann funktionieren, wenn Aufrufe von in der Vorlage deklarierten Methoden (auch von solchen, welche von Ihnen implementiert werden) durch andere, korrekte Implementationen ersetzt werden.

Der Verstoß gegen verbindliche Anforderungen führt zu Punktabzügen und kann die korrekte Bewertung Ihrer Abgabe unter Umständen beeinflussen. Die Implementation einer in der Quelltext-Vorlage deklarierten Methode wird nur bewertet, wenn der mit `TODO` markierte Exception-Wurf entfernt wird.

Hinweise für alle Hausübungen:

Die zu verwendenen Zugriffsmodifizierer sind in der Vorlage bereits gegeben und werden auf dem Übungsblatt nicht immer angegeben. Beachten Sie die Informationen im Moodle-Abschnitt *Technisches und Probe-Übungsblatt*.

Bei Fragen stehen wir Ihnen vorzugsweise im Moodle-Kurs und in den Sprechstunden zur Verfügung.

Die für diese Hausübung in der Vorlage relevanten Verzeichnisse sind `src/main/h04` und `src/test/h04`.

Hinweise (für dieses Übungsblatt zum Nachschlagen nützliche Quellen und erlaubte Konstrukte**):**

Java-Dokumentation: `Math`, `List`, `Random` `ArrayList`, `Collections`, `Iterator`, `ListIterator`, `Comparator`, `Duration`, `ThreadMXBean` und `h04.util.Permutations`
Bei Unklarheiten zu den einzelnen Aufgaben, können Sie ggf. weitere Informationen aus den JavaDocs der einzelnen Klassen/ Methoden entnehmen.

Einleitung

In diesem Übungsblatt beschäftigen wir uns mit hybriden Algorithmen, genauer gesagt hybride Sortieralgorithmen. Was ist überhaupt ein hybrider Algorithmus und wofür werden Sie verwendet?

Ein hybrider Algorithmus ist ein Algorithmus, der zwei oder mehr andere Algorithmen zur Lösung desselben Problems kombiniert. Je nach Datenlage oder im Verlauf eines Algorithmus wird zwischen ihnen gewechselt. Dies geschieht in der Regel, um gewünschte Eigenschaften der einzelnen Algorithmen zu kombinieren, so dass der Gesamtalgorithmus besser ist als die einzelnen Komponenten.

Auf diesem Übungsblatt implementieren Sie einen hybriden Sortieralgorithmus und bestimmen empirisch die optimale Sequenzgröße, ab der von einem Algorithmus zum anderen umgeschaltet werden sollte. Diese optimale Sequenzgröße ist allerdings nicht eine feste Konstante, sondern hängt vom Grad der Unsortiertheit (*disorder*) der Sequenz ab und wird von Ihnen empirisch bestimmt.

Wir verwenden die folgenden Algorithmen (1) *Selection Sort* und (2) *Merge Sort*.

Der hybride Sortieralgorithmus startet mit Merge Sort, da dieser eine bessere Komplexität besitzt. Sobald die optimale Sequenzgröße erreicht ist, schaltet Merge Sort zu Selection Sort um.

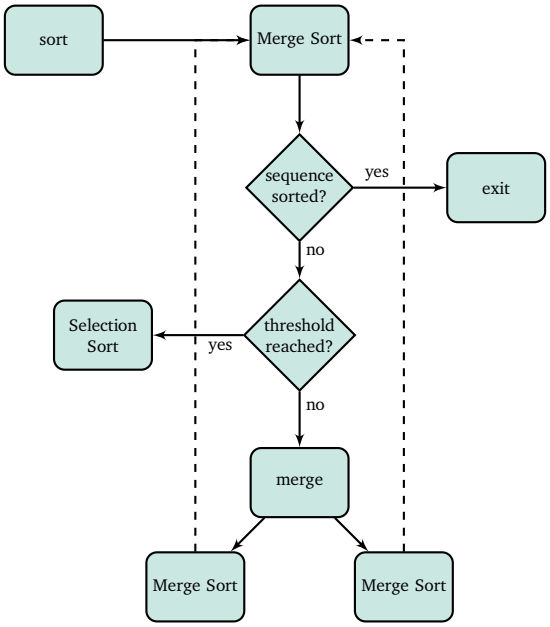


Abbildung 1: Abstrakte Visualisierung des hybriden Sortieralgorithmus

Verbindliche Anforderungen für die gesamte Hausübung:

(i) Zahlen werden nur gerundet, wenn es nötig ist. Es wird immer zur nächsten Ganzzahl gerundet, sofern nichts anderes angegeben ist. (d.h. 2.3 wird zu 2, 2.5 wird zu 3 und 2.7 zu 3)

(ii) Es dürfen keine neuen `ListItem`–Objekte erzeugt werden und das Attribut `key` darf nicht überschrieben werden, sofern dies nicht explizit für die Aufgabe erlaubt ist. Die Umordnung der Liste geschieht allein durch Überschreiben von `next`-Attributwerten.

(iii) In einer `java.util.List` dürfen keine Elemente entfernt oder hinzugefügt werden, sofern dies nicht explizit für die Aufgabe erlaubt ist.

H1: Degree of Disorder — 6 Punkte

Um eine optimale Sequenzgröße zu bestimmen, müssen wir uns ein paar Funktionen zur Berechnung der Unsortiertheit der Sequenz definieren. Dazu definieren wir verschiedene Funktionen im Package `h04.function`, die die Unsortiertheit der Sequenz berechnen.

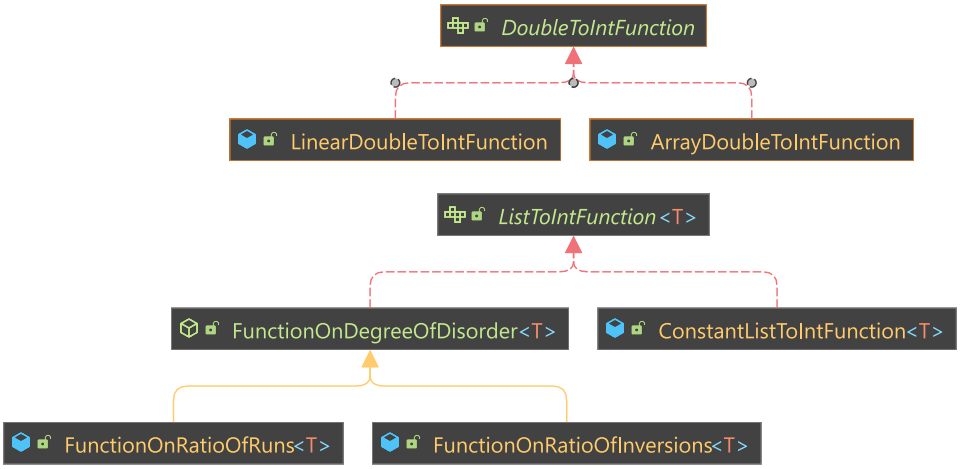


Abbildung 2: Übersicht der Funktionen

H1.1: DoubleToIntFunction — 2 Punkte

Analog zum funktionalen Interface `Function<T,R>` hat das nicht generisches Interface `DoubleToIntFunction` eine funktionale Methode `apply`, die einen Parameter vom formalen Typ `double` und Rückgabetyp `int` hat. Hierbei soll der Wert des aktualen Parameters nicht kleiner als 0.0 oder größer als 1.0 sein. Sollte dies dennoch der Fall sein, so soll jede Implementation von `apply` eine `IllegalArgumentException` werfen.

Aus dieser Funktion werden zwei Funktionen erzeugt:

(1) `ArrayDoubleToIntFunction`: Sei n die Länge des Arrays.

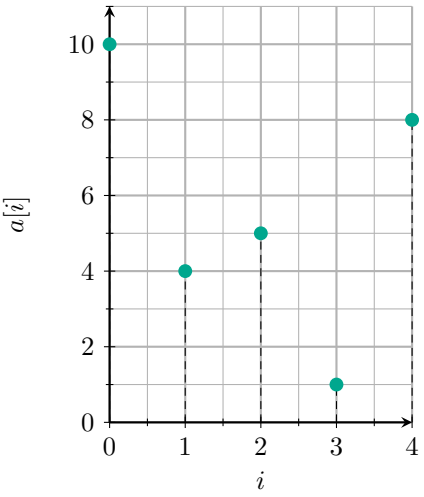
Für einen aktualen Parameter $x \in [0 \dots 1]$ soll das Ergebnis von `apply` wie folgt sein:

(a) Falls $x \cdot (n-1)$ nur maximal um 10^{-6} von einer ganzen Zahl $i \in \{0, \dots, n-1\}$ abweicht, soll `apply` den Wert im Array an Index i zurückliefern.

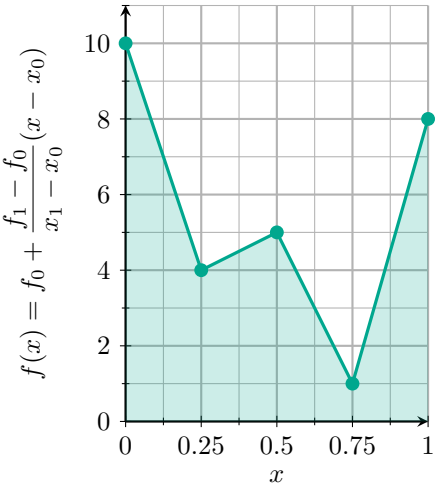
(b) Ansonsten soll in `apply` zuerst die lineare Interpolation [1] an x aus den beiden Werten des Arrays an den Indizes $\lfloor x \cdot (n-1) \rfloor$ und $\lceil x \cdot (n-1) \rceil$ berechnet und das auf diese Weise ermittelte Zwischenergebnis noch auf eine Ganzzahl gerundet[2] werden.

$$f(x) = f_0 + \frac{f_1 - f_0}{x_1 - x_0}(x - x_0)$$

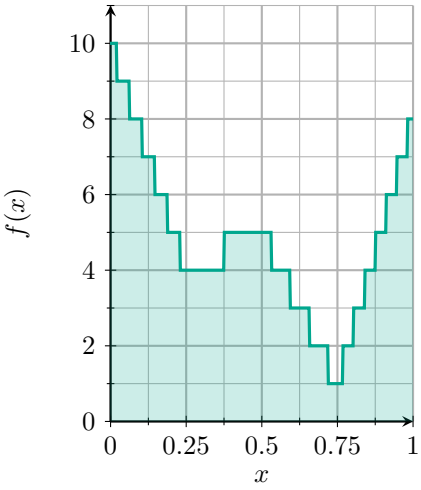
[1] `https://de.wikipedia.org/wiki/Interpolation_(Mathematik)#Lineare_Interpolation`
[2] `https://de.wikipedia.org/wiki/Abrundungsfunktion_und_Aufrundungsfunktion`



(a) Eingabe $a = \{10, 4, 5, 1, 8\}$



(b) Zwischenergebnis vor Rundung



(c) Ergebnis nach Rundung

(2) `LinearDoubleToIntFunction`: Anstelle eines Arrays besitzt die Klasse zwei Attribute und realisiert die Funktion $x \longrightarrow \text{round}(a \cdot x + b)$.

H1.2: ListToIntFunction — 4 Punkte

Mit der obigen Grundlage können wir Funktionen definieren, die auf einer Sequenz angewendet werden können, um eine optimale Sequenzgröße zu bestimmen. Das funktionale Interface `ListToIntFunction` mit generischem Typparameter `T` ermöglicht uns dies.

Wie bereits in der Einleitung erwähnt, wollen wir den Grad der Unsortierheit herausfinden. Dazu definieren wir zwei Algorithmen:

(1) Run: In Bezug auf eine gegebene Sortierlogik ist ein *Run* in einer nichtleeren Sequenz eine inklusionsmaximale Teilsequenz, die in sich aufsteigend sortiert ist. Dabei verstehen wir eine Teilsequenz genau dann als ***nicht*** inklusionsmaximal, wenn unmittelbar vor der Teilsequenz oder unmittelbar nach der Teilsequenz (oder beides) noch ein Element in der Liste ist und dieses zur Teilsequenz hinzugefügt werden kann, ohne dass dadurch die aufsteigende Sortierung verletzt wird. Insbesondere ist eine Sequenz die Konkatenation ihrer Runs.

Implementieren Sie diesen Algorithmus in der Klasse `FunctionOnRatioOfRuns`.

> **Beispiel:**
>
> Sequenz $S = (1, 2, 3, 4, 5 \,|\, 3, 4, 5, 5 \,|\, 4 \,|\, 3 \,|\, 2, 2, 2, 5, 6, 6, 7 \,|\, 6, 8, 9 \,|\, 5 \,|\, 1, 2, 3, 4, 5)$
> In der Sequenz S sind die Grenzen zwischen aufeinanderfolgenden Runs durch senkrechte Striche anstelle von Kommas angezeigt. Die Sequenz S weist 8 Runs auf.

(2) Inversion: Eine *Inversion* in einer Sequenz $(a_1, \ldots, a_n)$ ist dabei ein Paar (i, j) mit $1 \leq i < j \leq n$ und $a_i > a_j$.

Zur Normalisierung verwenden Sie hier nicht die Länge der Sequenz, sondern die maximal mögliche Anzahl an Inversionen. Somit wird gewährleistet, dass auch dieser Quotient wieder im Intervall $[0 \ldots 1]$ liegt.

Implementieren Sie diesen Algorithmus in der Klasse `FunctionOnRatioOfInversions`.

> **Beispiel:**
>
> Sequenz $S = (1, 2, 3, 4, 5, 3, 4, 5, 5, 4, 3, 2, 2, 2, 5, 6, 6, 7, 6, 8, 9, 5, 1, 2, 3, 4, 5)$
> Die Sequenz S weist 115 Inversions auf.

H2: Eigene Collections–Klasse mit eigener Methode sort 18 Punkte

Nun kommen wir zu der eigentlichen Aufgabe – Hybrider Sortieralgorithmus. Im Folgenden ist mit `ListItem` die generische Klasse aus Kapitel 07, ab Folie 116 der FOP gemeint. Analog zu Methode `sort` von `java.util.Collections` betrachten wir Sequenzen von beliebigem Referenztyp `T`, die als Objekte von java.util.List<T> gegeben sind, wobei die Sortierlogik durch java.util.Comparator<T> flexibel gehalten wird.

Die Klasse `MyCollections` soll die hybride Sortierung umsetzen und besitzt folgende Attribute, die bei der Implementierung hilfreich sind:

Bezeichner	Typ	Beschreibung
`function`	`ListToIntFunction<T>`	Berechnet die „Umschaltlänge“ für die zu sortierende Sequenz
`comparator`	`Comparator<? super T>`	Ermöglicht das Vergleichen von Elementen in einer Sequenz

Tabelle 1: Übersicht der Attribute

H2.1: sort(List) 2 Punkte

Implementieren Sie die Einstiegsmethode für die Sortierung einer Sequenz.

Die Methode `sort` erstellt eine Liste aus Objekten von `ListItem<T>` mit denselben Schlüsselwerten in derselben Reihenfolge wie in ihrem aktualen Parameter. Damit ruft sie `adaptiveMergeSortInPlace` aus Aufgabe H2.2 weiter unten auf und baut aus deren Rückgabe ihren aktualen Parameter neu auf, so dass er also dieselben Elemente, aber nun in sortierter Reihenfolge hat, indem die unsortierte Liste zu einer sortierten Liste geordnet wird. Mit anderen Worten: Nach außen hin arbeitet `sort` mit dem aktualen Parameter, intern aber mit einer Kopie in Form von `ListItem<T>`. Der zweite aktuale Parameter für den Aufruf von `adaptiveMergeSortInPlace` kommt aus dem `ListToIntFunction<T>`-Objekt mittels Methode `apply`.

Dazu sollen Sie zusätzliche folgende Methoden implementieren:

Bezeichner	Beschreibung
`listToListItem`	Konvertiert eine `List<T>` zu einer `ListItem<T>`, indem neue `ListItem<T>`–Objekte erzeugt werden.
`listItemToList`	Überschreibt die Elemente von einer `List<T>` mit den Schlüsselwerten aus den `ListItem<T>`–Objekten

Tabelle 2: Hilfsmethoden für `sort(List)`

Hinweis:

Beachten Sie die Anforderung, dass die Methode `Collections#sort(List, Comparator)` und damit auch die Methode `MyCollections.sort(List)` stabil sein soll (siehe dazu Kapitel 12, ab Folie 182 der FOP).

Anmerkung:

Bei diesem Beispiel kosten die Transformationen `java.util.List` $\longleftrightarrow$ `ListItem` mutmaßlich einen erheblichen Anteil der Gesamtlaufzeit. Daher ist in diesem konkreten Fall fraglich, ob die Laufzeit nicht besser wäre, wenn die Algorithmen direkt auf `java.util.List` arbeiten würden. Aber in vielen anderen Kontexten ist die Laufzeit des eigentlichen Algorithmus so hoch im Vergleich zu einer solchen Transformation und auch der Laufzeitgewinn durch Transformation auf eine für den Algorithmus effizientere Datenstruktur so hoch, dass es sich absolut lohnt.

Unbewertete Verständnisfragen:

(1) Warum muss unsere Methode `MyCollections#sort(List)` eine Obektmethode sein, während die Methode `java.util.Comparator#sort(java.util.List,java.util.Comparator)` eine Klassenmethode ist? Was spricht für Objekt-, was für Klassenmethode?

(2) Warum wird in der Methode `MyCollections#sort(List)` der `Comparator` nicht wie in der Methode `Collections.sort(List, Comparator)` als zweiter Parameter eingebracht, was wäre ein mögliches Problem?

(3) Die Methode `compare` von Interface `Comparator` ist eine Objektmethode. Würde nicht auch eine Klassenmethode ausreichen? Anders herum gefragt: Können Sie sich Situationen vorstellen, in denen der Zustand des `java.util.Comparator`–Objekts den Vergleich beeinflussen sollte? Falls Ihnen nichts einfällt, schauen Sie sich nochmals die Beispiele in Kapitel 06, Folien 103-137 der FOP an.

H2.2: MergeSort — 7 Punkte

Wir implementieren nun den ersten Sortieralgorithmus. Implementieren Sie in der Klasse `MyCollections` die Methode `adaptiveMergeSortInPlace` mit einem Parameter vom formalen Typ `ListItem<T>` und einem Parameter vom formalen Typ `int`. Rückgabetyp ist `ListItem<T>`. Analog zu `selectionSortInPlace` liefert `adaptiveMergeSortInPlace` einen Verweis auf den Kopf der sortierten Liste zurück.

Die Methode `adaptiveMergeSortInPlace` implementiert eine Variante des Algorithmus Mergesort unter Einbeziehung von Selection Sort, Details weiter unten.

Die zu implementierende Variante von Mergesort unterscheidet sich von der Variante in der Vorlesung in zwei Punkten:

(1) Die Implementierung geht bei der Zerlegung der Eingabeliste in zwei Teillisten anders vor.

(2) Die Implementierung wird ab einer gewissen Sequenzlänge auf Selection Sort „umschalten“. Details zu beiden Punkten im Folgenden.

Verbindliche Anforderungen:

(i) Die Teillisten werden durch die Methode `split` aufgeteilt.

(ii) Die beiden Teillisten werden durch rekursiven Aufruf von `adaptiveMergeSortInPlace` sortiert.

(iii) Der Algorithmus ist ebenfalls stabil.

(iv) Für die Zerlegung einer Sequenz in zwei Teilsequenzen suchen Sie die „mittigste“ Grenze zwischen zwei Runs, das heißt: Sie zerlegen die Sequenz an der Grenze zwischen zwei Runs, und unter allen infrage kommenden Positionen wählen Sie eine, bei der die Längen der beiden Teilsequenzen sich am minimalsten unterscheiden (falls es zwei solche Positionen gibt, können Sie eine davon frei wählen).

Schreiben Sie die Zerlegung in der Methode `split`.

Beispiel:

In der Präambel dieses Übungsblattes wurde das Beispiel

$$S = (1, 2, 3, 4, 5 \,|\, 3, 4, 5, 5 \,|\, 4 \,|\, 3 \,|\, 2, 2, 2, 5, 6, 6, 7 \,|\, 6, 8, 9 \,|\, 5 \,|\, 1, 2, 3, 4, 5)$$

zur Einführung von Runs verwendet. Die beiden Teilsequenzen nach der Zerlegung sind

$$S_1 = (1, 2, 3, 4, 5 \,|\, 3, 4, 5, 5 \,|\, 4 \,|\, 3)$$
$$S_2 = (2, 2, 2, 5, 6, 6, 7 \,|\, 6, 8, 9 \,|\, 5 \,|\, 1, 2, 3, 4, 5).$$

(v) Die Rekursion von `adaptiveMergeSortInPlace` bricht ab, wenn mindestens einer der beiden folgenden Fälle eintritt:

(1) Die Länge der Sequenz ist kleiner oder gleich dem aktualen Wert des zweiten Parameters von der Methode `adaptiveMergeSortInPlace`.

(2) Die Sequenz ist sortiert. Im Falle, dass (1), aber nicht (2) eintritt, wird `selectionSortInPlace` mit der Sequenz aufgerufen.

(3) Falls weder (1) noch (2) eintritt, werden die beiden Teilsequenzen durch je einen rekursiven Aufruf von `adaptiveMergeSortInPlace` mit jeder der beiden Teilsequenzen in sich sortiert, und die beiden sortierten Teilsequenzen werden wie in der Vorlesung mit dem Merge-Algorithmus `merge`, den Sie ebenfalls zu implementieren haben, zu einer sortierten Teilsequenz zusammengefügt.

H2.3: Selection Sort — 9 Punkte

Zum Schluss schreiben Sie den zweiten Sortieralgorithmus.

Implementieren Sie dazu die Methode namens `selectionSortInPlace`. Die Methode `selectionSortInPlace` hat einen Parameter vom formalen Typ `ListItem<T>` und liefert `ListItem<T>` zurück.

Der Algorithmus geht im Prinzip so wie der in Vorlesung AuD vor, siehe auch Kapitel 12, ab Folie 298 der FOP. Die äußere und die innere Schleife von Selection Sort haben hier auch dieselbe Variante und Invariante wie in diesen beiden Vorlesungen (inklusive Stabilität des Sortieralgorithmus). Der wesentliche Unterschied tritt in jedem Durchlauf der äußeren Schleife nach Termination der inneren Schleife auf: Das Listenelement, das in der inneren Schleife als das maximale im noch unsortierten Teil der Liste identifiziert wurde, wird nicht mit einem anderen Element vertauscht, sondern aus der Liste ausgekoppelt und an der passenden Position wieder eingekoppelt. (Überschreiben des `next`–Attributes)

Hinweis:

Beachten Sie, dass die ganzen Zusatzoperationen, die in Kapitel 12, ab Folie 298 der FOP für die Stabilität von Selection Sort notwendig waren, hier ersatzlos entfallen können.

Unbewertete Verständnisfrage:

Warum kann bei `selectionSortInPlace` nicht wie bei `sort` in H2.1 einfach der erste aktuale Parameter modifiziert werden? Mit anderen Worten: Warum muss hier stattdessen das Ergebnis mit `return` zurückgeliefert werden? Warum ist das bei `sort` in H2.1 hingegen nicht notwendig?

H3: Bestimmung der optimalen Umschaltlänge von Mergesort auf Selection Sort 8 Punkte

In dieser Aufgabe werden Sie versuchen, für jede Kombination von der Methode `FunctionOnRatioOfRuns` bzw. `FunctionOnRatioOfInversions` und jeden Subtyp von `IntToDoubleFunction` eine Umschaltstrategie zu finden, für die die zum Sortieren einer n-elementigen Liste verwendete CPU-Zeit minimal ist.

Hierfür werden Sie im späteren Verlauf der Aufgabe zufällige Permutationen unter der Verwendung von festen Umschaltlängen sortieren und die benötigte CPU-Zeit messen. Für jede Anzahl an Runs bzw. Inversionen speichern Sie dann diejenige Umschaltlänge, für die die Dauer am niedrigsten war. Da es den Rahmen sprengen würde, diesen Algorithmus so lange anzuwenden, bis für jede einzelne Anzahl an Runs bzw. Inversionen genügend Umschaltlängen ausprobiert wurden, werden Sie die beiden Kenngrößen in eine feste Menge sortierter Bins gruppieren. Stattdessen suchen Sie dann die optimale Umschaltlänge für jeden solchen Bin; hierzu später mehr.

H3.1: DoubleToIntFunctionFitter 4 Punkte

Zunächst benötigen wir einen Weg, eine `DoubleToIntFunction` aus der möglicherweise unvollständigen Menge an optimalen Umschaltlängen zu konstruieren. Dies erfolgt mittels dem Interface `DoubleToIntFunctionFitter` im Package `h04.function`. Das Interface hat eine nicht implementierte Methode `fitFunction` mit Rückgabewert vom Typ `DoubleToIntFunction`, die ein Array `y` vom Komponententyp `Integer` als formalen Parameter erwartet.

Dieses soll die Funktionswerte für eine Menge äquidistanter x-Werte aus dem Intervall $[0 \dots 1]$ enthalten. Angenommen die Funktion, aus der die Werte in `y` entstammen, heißt f, dann soll der Wert `y[i]` also $f(\frac{i}{\texttt{y.length}-1})$ entsprechen. Hierbei soll es möglich sein, dass der Funktionswert an manchen Stellen auch unbekannt ist. In diesem Fall enthält `y` an diesen Stellen den Wert `null`. Sie dürfen jedoch davon ausgehen, dass das erste und letzte Element von `y` niemals `null` sind.

Implementieren Sie nun zwei Funktionen (Klassen) namens `LinearInterpolation` und `LinearRegression`, die beide jeweils `DoubleToIntFunctionFitter` implementieren.

Die Implementation von `fitFunction` in `LinearInterpolation` soll dabei wie folgt vorgehen:

(1) Zunächst wird ein neues Array vom Komponententyp `double` mit derselben Länge wie `y` erzeugt, dessen Elemente an allen Stellen, an denen `y` nicht den Wert `null` enthält, identisch mit denen aus `y` sind.

(2) Für alle Abschnitte, an denen die Komponenten von `y` durchgehend `null` sind, sollen die Werte im `double`-Array linear aus denjenigen Elementen am jeweils nächstkleineren und nächstgrößeren Index, an denen die Funktionswerte bekannt sind, interpoliert werden.

(3) Weiterhin soll ein neues Array vom Komponententyp `int` mit derselben Länge wie `y` erzeugt werden, das die gerundeten Werte aus dem `double`-Array enthält.

(4) Schließlich liefert die Methode als Rückgabe ein Objekt vom Typ `ArrayDoubleToIntFunction`, das mit dem `int`-Array initialisiert wurde.

Sollte es Ihnen gelingen, dürfen Sie für eine bessere Performanz selbstverständlich gern die Stichpunkte (2) und (3) zu einem Schritt zu kombinieren.

Für die Implementation von `fitFunction` in `LinearRegression` führen Sie eine sogenannte lineare Regression durch um eine `LinearDoubleToIntFunction` zu bestimmen, die die gegebenen Funktionswerte möglichst gut beschreibt. Eine auf diese Weise bestimmte Funktion nennt man auch eine Ausgleichsgerade. Hierfür betrachten Sie jeden Funktionswert zusammen mit seinem zugehörigen x-Wert als Datenpunkt. Datenpunkte sind nur an den Stellen vorhanden, an denen `y` nicht den Wert `null` enthält. Wie oben bereits erwähnt, berechnet sich der x-Wert für einen Funktionswert in `y` am Index `i` durch `i / (y.length-1)`. Für eine Menge $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ von n Datenpunkten berechnet sich die Steigung β_1 der Ausgleichsgeraden dann wie folgt:

$$\beta_1 = \frac{\sum_{i=1}^{n} (x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

Dabei sind $\bar{x}$ und $\bar{y}$ die Durchschnitte über die x- bzw. y-Werte:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i, \qquad \bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$$

Schließlich berechnet sich der y-Achsenabschnitt β_2 der Ausgleichsgerade durch

$$\beta_2 = \bar{y} - \beta_1 \bar{x}$$

Als Ergebnis von

`fitFunction` liefern Sie schließlich ein Objekt vom Typ `LinearDoubleToIntFunction` zurück, das sie mit den Werten von β_1 und β_2 initialisieren.

H3.2: SortingExperiment — 4 Punkte

Schließlich können wir mit dem Sorting Experimennt anfangen, um eine optimale Umschaltlänge empirisch zu bestimmen. Des erfolgt mit der Methode `computeOptimalThresholds` mit Rückgabetyp `Integer[][]`. Die Methode besitzt folgende Parameter:

Parameter	Datentyp	Beschreibung
`n`	`int`	Die Länge der Liste, die sortiert werden soll, um die CPU-Zeit des Algorithmus zu messen
`swaps`	`int`	Die maximale Anzahl an Vertauschungen, die durchgeführt werden soll um die zufälligen Permutationen zu erzeugen
`bins`	`int`	Die Anzahl an Bins, in denen die Kennzahlen von Runs und Inversionen jeweils gruppiert werden sollen
`gamma`	`double`	Der Mindestanteil der Umschaltlängen, der für einen Bin ausprobiert werden soll, um ein gültiges Ergebnis zu liefern (mehr dazu später)

Tabelle 3: Übersicht der Parameter

Das Ergebnis der Methode `computeOptimalThresholds` soll ein zweidimensionales `Integer`–Array mit der Form `new Integer[2][bins]` sein. Hier steht der erste Index für die Kennzahl (0 für Runs, 1 für Inversions) und der zweite für den jeweiligen Bin. Die Werte des Arrays sollen die Umschaltlängen sein, für die der Sortiervorgang unter den entsprechenden Gegebenheiten die geringste CPU-Zeit in Anspruch genommen hat.

Beispiel:

Angenommen Sie arbeiten auf einer 10-elementigen Permutation und verwenden 5 Bins, dann sollte der Rückgabewert von `computeOptimalThresholds` am Index `[0][2]` beispielsweise die Umschaltlänge liefern, mit der das Sortieren aller Permutation mit 5 oder 6 Runs am schnellsten vonstatten ging. Im selben Beispiel wäre der Wert am Index `[1][4]` die Umschaltlänge, mit der der Algorithmus für alle Permutation mit 37–45 Inversionen die geringste CPU-Zeit in Anspruch genommen hat.

Sei p eine Sequenz, führen Sie folgende Schritte insgesamt zweimal durch, wobei Sie p im ersten Durchlauf jedes Mal mit $[1, 2, 3, \ldots, n]$ initialisieren und im zweiten mit $[n, n-1, n-2, \ldots, 1]$.

Im Allgemeinen berechnen Sie den Index $k_{\text{run}}(r)$ desjenigen Bins, in dem sie die optimale Umschaltlänge für das Sortieren einer Liste mit r Runs einsortieren, durch

$$k_{\text{run}}(r) = \left\lfloor \frac{\texttt{bins} \cdot (r-1)}{\texttt{n}} \right\rfloor$$

Analog ergibt sich der Index $k_{\text{inv}}(i)$ der entsprechenden Bin für eine Liste mit i Inversionen durch

$$k_{\text{inv}}(i) = \left\lfloor \frac{\texttt{bins} \cdot i}{\frac{\texttt{n} \cdot (\texttt{n}-1)}{2} + 1} \right\rfloor$$

Die Indizes beginnen hier jeweils mit 0, und $\lfloor \cdot \rfloor$ steht für die Abrundungsoperation.

Zur Bestimmung der optimalen Umschaltlängen gehen Sie nun wie folgt vor: Sie beginnen mit einer Liste p der Länge n als Objekt vom Typ `List<Integer>`. Für jede Kombination aus Umschaltlänge $u \in \left\{n, \left\lceil \frac{n}{2} \right\rceil, \left\lceil \frac{n}{4} \right\rceil, \left\lceil \frac{n}{8} \right\rceil, \left\lceil \frac{n}{16} \right\rceil, \ldots, 1\right\}$ und Anzahl an Vertauschungen $s \in \{0, 1, 2, \ldots,$ `swaps`$\}$ bestimmen sie zunächst sukzessive s Paare von Listenindizes i und j mit $i \neq j$ und vertauschen die entsprechenden Elemente. Hierfür bietet sich die Verwendung von `java.util.Collections.swap` an.

Durch die s Vertauschungen ergibt sich eine Liste p'. Beachten Sie, dass aus $s = 0$ immer $p = p'$ folgt. Im nächsten Schritt erzeugen Sie ein neues Objekt vom Typ `MyCollections`, das Sie mit einer `ConstantListToIntFunction` mit der Konstante u initialisieren. Als `Comparator` übergeben Sie die natürliche Ordnung des Typen `Integer`. Dieses `Comparator`-Objekt verwenden Sie, um die Liste p' aufsteigend zu sortieren. Messen Sie die insgesamt für diesen Aufruf benötige CPU-Zeit (siehe Kapitel 13, ab Folie 39 der FOP).

Verwenden Sie anschließend eine `assert`-Anweisung um zu überprüfen, dass es sich bei der resultierenden Liste tatsächlich um eine sortierte Permutation von p handelt.

Sollte für den entsprechenden Bin einer der Kennzahlen Runs oder Inversionen von p bisher noch kein Messwert vorgekommen sein oder der neue Messwert besser sein als alle vorherigen, so speichern Sie diesen zusammen mit der verwendeten Umschaltlänge u. Im Falle eines Gleichstandes bei den Messwerten entscheiden Sie sich für die kleinere Umschaltlänge.

Vergessen Sie nicht, p bzw. p' im Anschluss wieder neu zu initialisieren!

Es ist möglich, dass Sie bei diesem Vorgehen für manche Bins nur sehr wenige Messwerte erlangen und selbst die Umschaltlänge mit der geringsten CPU-Zeit weit vom Optimum entfernt ist. Speichern Sie daher in einem dreidimensionalen Array vom Typ `boolean`, für welche Bins Sie bereits welche Umschaltlängen ausprobiert haben. Der erste Index soll hier die Kennzahl (Runs oder Inversionen) angeben und der zweite den entsprechenden Bin für diese Kennzahl. Der dritte Index steht für die Umschaltlänge, und der Wert der Komponente soll genau dann `true` sein, wenn die Umschaltlänge für diese Kombination aus Kennzahl und Bin bereits vorkam.

Setzen Sie im Rückgabe-Array von `computeOptimalThresholds` alle Komponenten auf `null`, für die nicht mindestens $\frac{\text{gamma} \cdot \lceil \log n \rceil}{\log 2}$ unterschiedliche Umschaltlängen getestet wurden.

Erweitern Sie die Klasse `SortingExperiment` schließlich noch um eine `main`-Methode, in der Sie die optimalen Umschaltlängen für eine Liste mit 1000 Elementen bestimmen. Verwenden Sie 800 als maximale Anzahl an Vertauschungen und gruppieren Sie die Kennzahlen jeweils in 100 Bins. Für `gamma` übergeben Sie 0.5. Nutzen Sie die resultierenden Arrays von Umschaltlängen, um mittels Objekten der Typen `LinearRegression` und `LinearInterpolation` insgesamt vier optimale `ListToIntFunction`-Objekte zu erzeugen und bestimmen Sie für alle vier die durchschnittliche Sortierzeit auf 500 zufällig erzeugten Permutationen von $[1, 2, 3, \ldots, 1000]$.

Geben Sie anschließend die Ergebnisse in der Konsole im folgenden Format aus:

```
"{1}: average elapsed time: {2}ms"
```

`{1}` entspricht hierbei `"Linear regression "`/`"Linear interpolation "` konkateniert mit `"ratio of runs"`/`"ratio of inversions"` und `{2}` entspricht die auf zwei Nachkommastellen gerundete durchschnittlich gemessene Zeit.

Unbewertete Verständnisfragen:

(1) Wie schneiden die Zeiten im Vergleich zueinander ab? Wie viel besser sind sie im Vergleich zum reinen Mergesort bzw. Selection Sort?

(2) Warum reicht es prinzipiell, einen Sortieralgorithmus wie Mergesort oder Selection Sort, der auf paarweisen Vergleichen beruht, allein mit Permutationen von $\{1, \ldots, n\}$ auf Korrektheit prüfen?

(3) Haben wir einen Fall übersehen für die Korrektheit des Algorithmus?