Übungsblatt 4

Abgabe: 11.06.2023

Auf diesem Übungsblatt geht es um Varianten von Sortierverfahren, die in der Vorlesung vorgestellt wurden. Diesmal macht die Person die Implementierung, die in einem Telefonbuch zuletzt gelistet würde. Die andere Person macht die Tests.

Aufgabe 1 Schneller einfügen

Beim Sortieren durch Einfügen, das in der Vorlesung vorgestellt wurde, wurde bei der Suche nach der Einfügestelle auch gleich der Platz für das einzufügende Element geschaffen. Es kann aber auch sinnvoll sein, die Suche nach der Einfügestelle vom Schaffen des Platzes zu trennen. Ersteres ist recht langsam, weil für jeden Vergleich die *compareTo*- bzw. *compare*-Methode aufgerufen werden muss, während das Verschieben von Speicher vergleichsweise schnell geht. Da der Teil des Arrays, in dem nach der Einfügestelle gesucht wird, bereits sortiert ist, bietet es sich an, hierfür eine binäre Suche zu verwenden, während das Verschieben der Elemente mit *System.arraycopy* gemacht werden kann. Damit reduziert sich zwar nicht der Aufwand im Sinne des O-Kalküls, aber für nicht zu große Arrays kann das Sortieren so tatsächlich schneller als z.B. das Sortieren durch Mischen sein. Implementiert diesen Ansatz. Beachtet dabei, dass Sortieren durch Einfügen ein stabiles Sortierverfahren ist und auch bleiben soll.

Aufgabe 1.1 Wohin damit? (20 %)

Implementiert das Finden der Einfügestelle in der Methode *findInsertionPoint*.

Aufgabe 1.2 Aus dem Weg! (20 %)

Vervollständigt die Methode *insertionSort*, wobei ihr den Speicher mit der Methode *System.arraycopy* verschiebt.

Aufgabe 2 Schneller als schnell

Die in der Vorlesung vorgestellte Variante von Quicksort ist tatsächlich nur eine von vielen Möglichkeiten, diesen Ansatz umzusetzen. So wurde z.B. beim Algorithmus aus der Vorlesung das Pivot-Element gesondert behandelt. Die Stelle, an der es im Array steht, wurde bei der Aufteilung in die zwei Teilfolgen ausgespart und das dort gespeicherte Element später an die Trennstelle der beiden Bereiche getauscht. Das Pivot-Element kann aber auch wie alle anderen Elemente behandelt werden, d.h. sein Platz im Array nimmt an der normalen Aufteilung teil. Die Aufteilung startet also bei *bottom* und nicht bei *bottom + 1*.

Aufgabe 2.1 Die Qual der Wahl (10 %)

Der Nachteil dieses Ansatzes ist, dass damit kein Element des Arrays benannt werden kann, das nach der Aufteilung bereits garantiert an der richtigen Stelle steht. Deshalb muss das Pivot-

Element so gewählt werden, dass es nicht das kleinste im zu sortierenden Bereich ist. Warum ist das so?

Aufgabe 2.2 Immer auf die Kleinen (20 %)

Implementiert die Methode *selectPivot*, die den zu sortierenden Bereich von unten durchsucht, bis ein Element gefunden wird, das sich vom ersten Element im Bereich unterscheidet. *selectPivot* liefert dann das größere von beiden zurück, wodurch das Pivot-Element niemals das kleinste ist. Sind alle Elemente in dem Bereich gleich groß, liefert *selectPivot null* zurück.

Aufgabe 2.3 Nicht über die Stränge schlagen (10 %)

Die beiden inneren Schleifen beim Quicksort aus der Vorlesung suchen nach Elementen, die in ihrem jeweiligen Teilbereich fehl am Platz sind (im Vergleich mit dem Pivot-Element) und deshalb vertauscht werden müssen. Beide Schleifen haben jeweils zwei Fortsetzungsbedingungen, nämlich dass sich der Index noch im gültigen Bereich befinden muss und dass das betrachtete Element bereits im richtigen Teilbereich ist. Sie lassen sich etwas beschleunigen, indem auf die Index-Bereichsprüfung verzichtet wird. Erläutert, warum dies mit der neuen Wahl des Pivot-Elements aus Aufgabe 2.2 nun möglich ist, ohne dass die Indizes den gültigen Bereich verlassen können.

Aufgabe 2.4 Gleichheit für alle (20 %)

Implementiert nun Quicksort so, dass es *selectPivot* benutzt und den Speicherort des Pivot-Elements im Array nicht mehr explizit behandelt. Verzichtet auf die Bereichsprüfung in den inneren Schleifen. Zudem gewinnt eure Implementierung eine neue Eigenschaft hinzu, denn *selectPivot* erkennt ja, wenn alle Elemente im zu sortierenden Bereich gleich sind und dieser somit bereits sortiert ist. Berücksichtigt dies in euer Implementierung.