technische universität
dortmund

Fakultät für Informatik
Lehrstuhl 11

Mart Hagedoorn
Guangping Li

Sommersemester 2024
12. April

DAP2 Praktikum – Blatt 2

Abgabe: W17

Studienleistung

- Zum Bestehen des Praktikums muss jeder Teilnehmer die folgenden Leistungen erbringen:
  - Es müssen mindestens 50 Prozent der Punkte in den Kurzaufgaben erreicht werden.
  - Es müssen mindestens 50 Prozent der Punkte in den Langaufgaben erreicht werden.
- Im Krankheitsfall kann ein Testat bei Vorlage eines Attests in der folgenden Woche nachgeholt werden.
- Wenn ein Praktikumstermin auf einen Feiertag fällt, müssen Sie sich an einem beliebigen anderen Praktikumstermin in der gleichen Woche testieren lassen.
- **Hinweis:** Notieren Sie sich Ihre Punkte nach jedem Testat! Dies dient der eigenen Kontrolle. (Ihr Punktestand kann Ihnen während des Semesters nicht genannt werden.)

Hinweis zur Verwendung der externen Bibliotheken

Da für die Praktikumsaufgaben prinzipiell keine zusätzlichen Java-Bibliotheken notwendig sind, ist die Benutzung solcher für die Bearbeitung der Aufgaben zunächst **nicht** gestattet. Sollten im weiteren Verlauf des Praktikums Bibliotheken notwendig werden, so wird dies explizit in den Aufgaben vorgegeben.

Langaufgabe 2.1: InsertionSort (3 Punkte)

Diese Aufgabe besteht aus mehreren Teilen. Bitte lesen Sie sich die Aufgabe vorher komplett durch und überprüfen Sie nach der Bearbeitung, ob Sie nichts vergessen haben. **Verändern sie die Codes nur in dem markierten Abschnitt!**

- Implementieren Sie eine Methode `public static void insertionSort(int[] array)`, die den Algorithmus InsertionSort aus der Vorlesung implementiert, das die Werte in einem Array in **aufsteigender** Reihenfolge sortiert.
- Implementieren Sie eine Methode `public static boolean isSorted(int[] array)`, welche überprüft, ob das Array `array` aufsteigend sortiert ist.
- Stellen Sie mittels `Assertions` die Korrektheit Ihres Programms sicher (siehe Hinweise zu *Assertions*).
- Schreiben Sie die `main`-Methode, die den Algorithmus wie folgt testet:
  - Das Programm empfängt eine Ganzzahl n und dann eine Liste L von n Ganzzahlen über Standard-In. Beispiel:
    `{ echo 100; seq 100; } | java B2A1.java`
    oder
    `{ echo 100; shuf -i 0-9999 -n 100; } | java B2A1.java`
    Sie können davon ausgehen, dass die Eingabe korrekt ist und müssen den Code nicht um diese Eingabe Robust gestalten.
  - Ergänzen Sie die `main`-Methode, so dass die Methode `insertionSort` das befüllte Array sortiert.
  - Zusätzlich soll die Anzahl der Vergleiche **zwischen zwei Elementen des Arrays** gezählt werden, die notwendig sind um das Array zu sortieren. Verwenden Sie dazu die Klassenvariable `compare_cnt`, die die Anzahl der benötigten Vergleiche speichert. Für jeden Vergleich von zwei Elementen in dem Array soll die Variable um 1 inkrementiert werden. Beachten Sie, dass auch bei bereits sortierten Arrays möglicherweise Vergleiche zwischen Elementen ausgeführt werden müssen.
  - Anschließend soll geprüft werden, ob das Array korrekt sortiert ist, und entsprechend `"Feld ist sortiert!"` bzw. `"FEHLER: Feld ist NICHT sortiert!"` ausgegeben werden.
  - Bei höchstens (inklusive) 100 Elementen soll in *einer neuen Zeile* das erzeugte unsortierte Feld und in *einer weiteren Zeile* das sortierte Feld ausgegeben werden. Die Elemente der Liste sollen durch Leerzeichen getrennt werden. Dabei muss die vorgegebene Reihenfolge eingehalten werden. Bsp. eines korrekten Aufrufes:

```
Input:  { echo 10; seq 10 -1 1; } | java B2A1.java
Output: 10 9 8 7 6 5 4 3 2 1
        1 2 3 4 5 6 7 8 9 10
        Feld ist sortiert!
        Das Sortieren des Arrays hat 45 Vergleiche benoetigt.
```

Langaufgabe 2.2: Teile & Herrsche mit MergeSort (4 Punkte)

Ergänzen Sie den Codegerüst für eine weitere Klasse `B2A2` in der Datei `B2A2.java`.

Die Methode `public static void mergeSort(int[] array)` erzeugt ein temporäres Hilfs-Array `tmpArray`, in dem Zwischenergebnisse des MergeSort-Algorithmus gespeichert werden sollen. Implementieren Sie nun in einer weiteren Methode `public static void mergeSort(int[] target, int[] source, int left, int right)` den rekursiven MergeSort-Algorithmus aus der Vorlesung. Aus Effizienzgründen erhält diese Implementierung neben dem zu sortierenden Feld `target` auch das bereits allozierte Feld `source`, in das temporäre Zwischenergebnisse von `mergeSort` geschrieben werden sollen. Darüber hinaus können Sie natürlich eigene Hilfsfunktionen schreiben.

Implementieren Sie die `main`-Methode wie in Aufgabe 2.1.

```
Aufruf mit: java B2A2.java  [size] [sequence]
Beispiel: { echo 100; seq 100; } | java B2A2.java
```

Der Test auf Sortiertheit und die Ausgaben haben wie in Aufgabe 2.1 zu erfolgen. Stellen Sie auch bei dieser Aufgabe mittels `Assertions` die Korrektheit Ihres Programms sicher (siehe Hinweise zu *Assertions*).

Die Ausgaben sollen analog zu Aufgabe 2.1 erfolgen, z.B.:

```
Input: { echo 6; seq 6 -1 1; } | java B2A2.java
Output: 6 5 4 3 2 1
        1 2 3 4 5 6
        Feld ist sortiert!
        Das Sortieren des Arrays hat 9 Vergleiche benoetigt.
```

Sie können und sollten Ihre Methoden, wie z.B. `InsertionSort.isSorted`, aus der erste Aufgabe wiederverwenden.

Beachten Sie die Hinweise und Tipps auf den folgenden Seiten.

Langaufgabe 2.3: Laufzeitmessung (1 Punkt)

Messen Sie die Laufzeit Ihrer Implementierungen mithilfe der Bibliotheken `java.time.Instant` und `java.time.Duration` (Beispiel unten). Sie sollten sicherstellen, dass *nur die zum Sortieren benötigte Zeit* gemessen wird, und nicht die Zeit zum Lesen der Eingabe! Überlegen Sie sich hierzu sinnvolle Testeingaben, und begründen Sie, warum Sie diese für sinnvoll halten, um die Laufzeiten von InsertionSort und MergeSort zu vergleichen. Versuchen Sie sich die beobachteten Laufzeiten auf der Grundlage Ihres Wissens aus der Vorlesung zu erklären.

```
Instant start = Instant.now();
// Code, dessen Laufzeit Sie messen wollen
// ...
Instant finish = Instant.now();
long time = Duration.between(start, finish).toMillis();
System.out.println("Time: " + time);
```

Hinweise und Tipps

Assertions

In Java gibt es das Konstrukt *Assertions*. Diese „Zusicherungen“ oder auch „Annahmen“ helfen dabei, Fehler im Programm zu finden, indem man z. B. Invarianten direkt im Code (in der Form von Assertions) mit angibt und diese dann zur Laufzeit überprüft werden. **Assertions sind nicht dazu gedacht um Benutzereingaben auf Korrektheit zu prüfen. Sie dienen z.B. dazu Schleifeinvarianten von Algorithmen zu prüfen, die innerhalb eines Schleiferumpfes gelten müssen, damit die Korrektheit des Algorithmus gegeben ist. Beispielsweise können wir bei dem InsertionSort immer davon ausgehen, dass ein sortiertes Teilarray bis zu einem bestimmten Index existiert.**

Syntax:

```
assert exp1;
```

oder:

```
assert exp1 : exp2;
```

Hierbei muss `exp1` immer ein boolscher Ausdruck sein. Ist dieser `true`, so ist die Assertion korrekt, also die „Annahme“ erfüllt. Falls `exp1` den Wert `false` annimmt, so kann mit `exp2` eine Fehlermeldung erzeugt werden, welche auf der Konsole ausgegeben wird.

Beispiel:

```
int laenge;
...
assert laenge > 0 : laenge + " ist nicht groesser 0";
int[] array = new int[laenge];
```

Bei laenge = -5 wäre die Fehlermeldung auf der Konsole dann: `-5 ist nicht groesser 0`

Um Assertions bei der Ausführung zu aktivieren, muss dem Interpreter der zusätzliche Parameter `-enableassertions` oder `-ea` mitgegeben werden.

Beispiele:

```
java -enableassertions Klasse
java -ea Klasse param1 param2 param3
```

Wichtig: Zur Messung von Laufzeiten sollten Assertions deaktiviert sein, da diese die Ergebnisse verfälschen können. Da Assertions explizit aktiviert werden müssen, sind sie im Allgemeinen kein geeignetes Mittel, um Benutzereingaben zu überprüfen.

Weitere Informationen zu Assertions finden sich unter:

`https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html`