Aufgabenblatt 2
Suchen und Sortieren

- Abgabetermin: **Samstag, 21.05.2016 23:55 Uhr**
- Zur Prüfungszulassung müssen in einem Aufgabenblatt mind. 25% der Punkte erreicht werden und alle weiteren Aufgabenblätter mit mindestens 50% der Punkte bestanden werden.
- Die Aufgaben müssen in *Zweiergruppen* bearbeitet werden.
- Jede Hausaufgabe enthält eine optionale Zusatzaufgabe, die bei erfolgreicher Bearbeitungen 1 Klausurpunkt zählt.
- Abgabe:
  - Die Abgabe erfolgt über CodeOcean[1]. CodeOcean ermöglicht die Bearbeitung und Ausführung der Quelldateien im Browser. Bei Bedarf können die Sources (Quelltexte) und Unit Tests von der Übungs-Webseite[2] heruntergeladen werden, um lokal zu entwickeln.
  - CodeOcean benutzt Java Version 8.
  - Alle Aufgaben sind über das CodeOcean einzureichen. CodeOcean ermöglicht das Anfügen von Kommentaren bei der Abgabe, darüber hinaus können auch Kommentare in den Source-Code geschrieben werden.
  - Geben Sie bei **jeder** Aufgabe beide Namen Ihrer Gruppe an.
  - Achten Sie darauf, dass Ihr Quelltext hinreichend kommentiert ist.

Aufgabe 1: Vergleiche und Sortieren in Java

5 P

Betrachten Sie die Klassen `assignment2.Student` und `assignment2.StudentManager`. Die `Student`-Klasse hält Daten über je einen Studenten, während die `StudentManager`-Klasse Informationen über eine Menge von Studenten-Objekten liefern. Bearbeiten Sie folgende Aufgaben:

a) In der Methode `nameExists((String firstName, String lastName, ArrayList<Student> students))` wird überprüft, ob in einer gegebenen Liste von Studenten-Objekten schon ein Student existiert, der den gegebenen Vor- und Nachnamen hat. Leider wurde diese Methode fehlerhaft implementiert. Finden und korrigieren Sie den Fehler.

b) Um eine Liste von Studenten nach deren Namen sortieren zu können implementiert `Student` das Interface `Comparable`. Hierzu benötigt es die Methode `public int compareTo(Student o)`. Implementieren Sie diese Methode so, dass Studenten lexikographisch sortiert werden – zuerst nach Nachnamen, dann nach Vornamen. Die Methode `ArrayList<Student> sortByName(ArrayList<Student> students)` im `StudentManager` nutzt die `compareTo`-Methode, um Studenten miteinander zu vergleichen.

c) Durch die Implementierung von `compareTo` können Studenten nach ihrem Namen geordnet werden. Das Studienreferat möchte jedoch die Studenten nach ihrer Matrikelnummer sortieren lassen, während die Übungsleiter daran Studenten nach der Verbesserung der Punkte sortieren möchten. Dies ist möglich mit Hilfe von **Komparatoren**, welche das Interface `Comparator` implementieren.

[1] `https://open.hpi.de/courses/pt2-2016`
[2] `https://hpi.de/naumann/teaching/current-courses/ss-16/uebung-programmierechnik-ii.html`

- Implementieren Sie den Komparator `assignmen2.StudentMatrNumberComparator`, der Studenten aufsteigend gemäß ihrer Matrikelnummer sortieren kann (dies heißt, dass eine kleine Matrikelnummer an vorderer Stelle kommt).
- Implementieren Sie den Komparator `assignmen2.StudentGradeImprovementComparator`, der Studenten gemäß ihrer Verbesserung von Übung zur Klausur sortiert. Studenten mit der größten Punkteverbesserung sollen an vorderster Stelle einsortiert werden.
- Implementieren Sie die Methode `ArrayList<Student> sortByMatrNumber(ArrayList<Student> students)` im `StudentManager`, der mittels des `assignmen2.StudentMatrNumberComparator` eine Liste von Studenten sortiert.
- Implementieren Sie die Methode `ArrayList<Student> sortByGradeImprovement(ArrayList<Student> students)` im `StudentManager`, der mittels des `assignmen2.StudentGradeImprovementComparator` eine Liste von Studenten sortiert.

d) Beantworten Sie in einem Quelltext-Kommentar folgende Frage für die Resultate von `sortByName`, `sortByMatrNumber` und `sortByGradeImprovement`:
Kann die resultierende sortierte Reihenfolge von Daten von der Sortierung der Eingabedaten abhängen oder ist sie für jede mögliche Permutation der Studenten-Objekte gleich? Treffen Sie geeignete Annahmen zur Verteilung der Attribute der Studenten-Objekte.

Aufgabe 2: Suchen

6 P

Die nachstehenden Aufgaben widmen sich dem Suchen spezieller Elemente ganzzahliger Arrays.

a) Implementieren Sie die Methode `findUnsorted(int x, int[] a)` der Klasse `assignment2.Search`, welche die Indizes sämtlicher Vorkommen von x innerhalb des **unsortierten** Arrays a in aufsteigender Sortierung zurückgibt. Das erwartete Resultat des Aufrufs $findUnsorted([5, 2, 4, 1, 4], 4)$ ist zum Beispiel $[2, 4]$.
Achten Sie auf eine effiziente Implementierung!

b) Implementieren Sie die Methode `findSorted(int x, int[] a)` der Klasse `assignment2.Search`, welche die Indizes sämtlicher Vorkommen von x innerhalb des **sortierten** Arrays a in aufsteigender Sortierung zurückgibt. Das erwartete Resultat des Aufrufs $findSorted([1, 2, 4, 4, 5], 4)$ ist zum Beispiel $[2, 3]$.
Achten Sie auf eine effiziente Implementierung!

c) Beobachten Sie die unterschiedlichen Laufzeiten der beiden Methoden für $|a| \geq 10^6$ und diskutieren Sie ihre Beobachtungen anhand eines Quelltextkommentars. Eine Datei mit vielen Daten steht beispielsweise hier bereit: `https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/lehre/SS2016/PTII_Uebung/large_array.zip`

Aufgabe 3: Sortieren

8 P

Die nachstehenden Aufgaben widmen sich dem Sortieren ganzzahliger Arrays.

a) Implementieren Sie die Methode `bubblesort(int[] a)` der Klasse `assignment2.Sort`, welche das Array a mittels *Bubblesort* aufsteigend sortiert. Der erwartete Inhalt des Arrays $a = [5, 2, 4, 1, 4]$ nach dem Aufruf *bubblesort* (a) ist zum Beispiel $[1, 2, 4, 4, 5]$.

b) Implementieren Sie die Methode `quicksort(int[] a)` der Klasse `assignment2.Sort`, welche das Array a mittels *Quicksort* aufsteigend sortiert. Der erwartete Inhalt des Arrays $a = [2, 3, 7, 2, 3]$ nach dem Aufruf *quicksort* (a) ist zum Beispiel $[2, 2, 3, 3, 7]$.

c) Implementieren Sie die Methode `countingsort(int[] a)` der Klasse `assignment2.Sort`, welche das Array a mittels *Countingsort* aufsteigend sortiert. Nehmen Sie an, dass $\forall i \in \mathbb{N}_0 \wedge i < |a| : a[i] \in \mathbb{N}_0$ gilt.

d) Beobachten Sie die unterschiedlichen Laufzeiten der drei Methoden für $|a| \geq 10^6$ und diskutieren Sie ihre Beobachtungen anhand eines Quelltextkommentars. Eine Datei mit vielen Daten steht beispielsweise hier bereit: `https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/lehre/SS2016/PTII_Uebung/large_array.zip`
Die statische Methode `sort(Object[] a)` der Klasse `java.util.Arrays` arbeitet abhängig von der spezifischen Eingabe gemäß *Mergesort* beziehungsweise *Quicksort*. Weshalb setzt jene Methode nicht auf *Countingsort*?

Zusatzaufgabe: Wortsuche

Gegeben sei ein Raster (zweidimensionales Array `char[][]`), in dem verschiedene Begriffe (`HashSet<String>`) versteckt sind. Begriffe können dabei horizontal und vertikal, sowie vor- und rückwärts auftauchen (also von links nach rechts, rechts nach links, oben nach unten und unten nach oben). Felder, die zu keinem Begriff gehören, sind mit anderen Buchstaben ausgefüllt.

Ihre Aufgabe ist es, algorithmisch die versteckten Begriffe zu finden, indem Sie die Methode `search` der Klasse `WordSearch` implementieren. Als Eingabe nimmt die Methode das ausgefüllte Raster und die Menge an Suchbegriffen. Zurückgegeben werden soll das Raster ohne die Füllzeichen (siehe Abbildung); ersetzen Sie diese durch den Bindestrich-Character (45).

```
MKLTQLPZKNPXCJJ          -----L---------
ZLNNSERRPGFRFPQ          -----E---------
ZFWTQGNSHJTJZRM          -----G--H------
KATZEOMLCFBVLXY          KATZEO--C------
VFWJJVRYSSNTHLB          -----V--S------
BMGPQDGQIPNRMNP          --------I-----P
GVBCKLZKFGJGJNF          --------F-----F
JCTTZLQTBNMTBRE    ➔     ---------------E
LJWQHDNVDBNDFLR          ---------------R
XWHNTJMYFKBGTVD          ------M-------D
DJKVBSADJXNVTXT          ------A--------
WBFQGDURWVDNUHS          ------U---DNUH-
KFKYKXSHXYMRXCK          ------S--------
KKMGVJXKDPWTFJR          ---------------
KYWVRYLLFPNLNPW          ---------------
```

Achten Sie auf einige Eigenarten des Rasters:

- Die Höhe und Breite des Rasters sind identisch; es ist also ein Quadrat.
- Das Raster arbeitet nur auf Großbuchstaben.
- Weiterhin enthält das Raster keine Sonderzeichen, Umlaute etc. (z. B. ist 'Bär' nicht erlaubt).
- Alle Suchbegriffe enthalten mindestens einen Vokal; die Füllzeichen sind ausschließlich Konsonanten.
- In jeder Zeile und jeder Spalte steht höchstens ein vollständiger Suchbegriff.
- Suchbegriffe überschneiden einander nicht. Sie kreuzen einander nicht.
- Kein Suchbegriff enthält einen anderen (z. B. sind 'Pferd' und 'Seepferdchen' zusammen nicht erlaubt).
- Jeder Suchbegriff ist genau einmal im Raster enthalten.
- Die Größe des Rasters und die Anzahl der Suchbegriffe können variieren (werden aber überschaubar groß bleiben).

Nutzen Sie bei Ihrer Implementierung die vorgenannten Hinweise um die Suche effizienter zu gestalten. Bewertet wird neben der Funktionsfähigkeit Ihrer Lösung auch, wie Sie zur Optimierung vorgegangen sind. Kommentieren Sie daher im Quellcode alle Annahmen die Sie treffen und alle Ansätze die Sie umsetzen.