Übungsblatt 5 – Lösungen

Aufgabe 1 - Quicksort

Sortiere die nachfolgenden Zahlenfolgen mit dem *Quicksort*-Algorithmus. Wähle als Pivotelement – wie in der Vorlesung vorgestellt – das letzte Element des zu sortierenden Bereichs. Gib die Werte der gewählten Pivotelemente in der Reihenfolge an, in der sie beim Sortieren bestimmt werden.

zu sortierenden Zahlenfolge: 7 15 3 17 19 10 5 6 11
Folge der Pivot-Elemente: 11 6 5 10 15 17

zu sortierenden Zahlenfolge: 3 5 6 7 10 11 15 17 19
Folge der Pivot-Elemente: 19 17 15 11 10 7 6 5

zu sortierenden Zahlenfolge: 19 17 15 11 10 7 6 5 3
Folge der Pivot-Elemente: 3 19 5 17 6 15 7 11

Aufgabe 2 - Sortieren nach Studienfächern

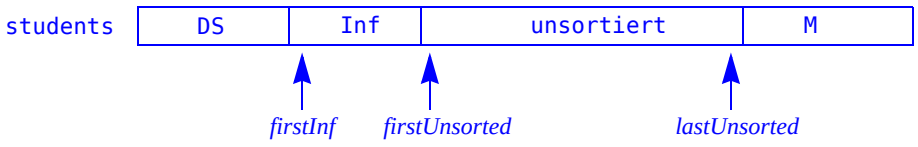
In der Vorlesung sind verschiedene Sortieralgorithmen am Beispiel der Klassen `Lecture` und `Student` eingeführt worden. Dabei wurden die Studierenden einer Vorlesung anhand ihrer Matrikelnummern oder anhand ihrer Namen sortiert. Da das Studienfach im Attribut `subject` ebenfalls als `String` abgelegt wird, könnten die bekannten Sortieralgorithmen beim Vorliegen geeigneter Vergleichsmethoden in der Klasse `Student` analog angepasst werden. Während jedoch jede Matrikelnummer nur genau einmal vorkommt und auch Namensgleichheiten eher selten sind, ist die Zahl der möglichen Studienfächer klein. Daher liegen beim Sortieren viele gleiche Studienfachangaben vor.

Nimm an, dass die Zahl der Studienfächer auf drei beschränkt sei – `DS`, `Inf`, `M` – für die folgende Ordnung gilt:

```
DS < Inf < M
```

Diese Einschränkung ermöglicht es, sehr schnell eine sortierte Reihenfolge herzustellen. Überlege und Skizziere, wie ein Algorithmus aussehen müsste, der das Sortieren von `Student`-Objekten nach ihren Studienfächern mit einer Laufzeit von $O(n)$ bewältigen könnte. Der Algorithmus soll *in-situ* arbeiten, also mit dem schon existierenden Feld `students` zur Verwaltung der `Student`-Objekte auskommen. Orientiere Dich an der Umsetzung der Vertauschung innerhalb des Quicksort-Algorithmus.

Die Lösungsidee besteht darin, nicht einzelne `Student`-Objekte zu betrachten, sondern ganze Bereiche von Objekten mit gleichem Studienfach. Dann ist es möglich, jedes Objekt nur einmal zu inspizieren und einem Bereich zuzuordnen.



Die Lösung zeigt ihre unmittelbare Verwandschaft zur Implementierung des Quicksort-Algorithmus. Die Lösung wird besonders einfach, wenn der unsortierte Bereich zwei der Gruppen trennt.

Voraussetzungfür eine $O(n)$-Lösung ist eine konstante Zahl von Studienfächern, die insbesondere nicht mit n wächst – sonst wären wir letztlich bei $O(n^2)$ o.ä. Bei der oben skizzierten Lösung werden Variablen für die *Grenzindizes* benötigt, deren Zahl bei der Implementierung festliegen muss.

Aufgabe 3 - Additions-/Subtraktionsausdrücke

Implementiere eine Methode, die prüft, ob eine Folge von Werten zu einem Ausdruck mit Additions- und Subtraktionsoperationen ergänzt werden kann, dessen Auswertung den Wert `0` ergibt.

Beispiele:

- Die Folge `5,3,2,4,1,0,3` kann zu folgendem Ausdruck mit Ergebnis `0` ergänzt werden: `5+3-2-4+1+0-3`.
- Die Folge `5,5,5` kann nicht zu einem passenden Ausdruck ergänzt werden.

Ein Aufruf der Methode `boolean addCalcExists(int[] values, int position, int result)` gibt `true` zurück, falls ein entsprechender Ausdruck existiert, sonst `false`.

Hinweis:
Die Parameter `position` und `result` sollen helfen, den erreichten Zwischenzustand und das erreichte Zwischenergebnis an die rekursiven Aufrufe weiterzugeben. Beim ersten Aufruf soll beiden Parametern der Wert `0` als Argument übergeben werden.

Erweiterung der Aufgabenstellung:
Erweitere die Methode so, dass sie die Folge der Operationen ausgibt, falls ein entsprechender Ausdruck möglich ist.
Ein Aufruf der Methode `String addCalcExp(int[] values, int position, int result, String exp)` gibt dann den Ausdruck zurück, sonst den Text `"calculation impossible"`.

Die Lösung durch einen rekursiven Backtracking-Algorithmus ist sehr einfach und setzt unmittelbar das Ausprobieren aller Additions- und Subtraktionskombinationen um.

Aufgabe 4 - Java Star

In der Vorlesung ist ein *Backtracking*-Algorithmus vorgestellt worden, mit dem festgestellt werden kann, ob ein Schiff, die *Java Star*, unter gewissen Randbedingungen mit vorgegebenen Containern beladen werden kann. Diese Randbedingungen sollen nun verschärft werden. Überlege Dir für die folgenden Anforderungen Algorithmen und implementiere diese als Java-Methoden:

- Die *Java Star* soll den Hafen nur dann verlassen dürfen, falls es sich nach dem Verladen des letzten Containers genau im Gleichgewicht befindet, also `divergence == 0` gilt. Implementiere dafür die Methode `existsTotalBalance`.

  Die Lösung ist sehr einfach, da der aus der Vorlesung bekannte Algorithmus strukturell nicht geändert werden muss. Es muss lediglich nach dem Verteilen aller Container überprüft werden, ob `balance == 0` gilt.

- Nach einem Umbau besitzt die *Java Star* nun drei Stauräume. Eine Ungleichheit in den Stauräumen an Backbord und Steuerbord führt zu der aus der Vorlesung bekannten Schieflage des Schiffes, ein Beladen des hinzugekommenen mittig liegenden Stauraums verändert die Lage des Schiffes nicht. Zugleich wird den drei Stauräume die gleiche Anzahl von Containern `containerLimit` zugeordnet, die maximal aufgenommen werden können. Implementiere eine entsprechende Methode `existsBalanceWith3Limited`.

  Die bekannte Lösung wird um die Alternative eines dritten Stauraums ergänzt. Alle Beladungen beginnen immer mit dem mittleren Stauraum, da dort das Gleichgewicht erhalten bleibt. Der Programmcode zu dieser Aufgabe enthält zusätzlich noch eine Lösung für eine beliebige Zahl von Stauräumen.

Aufgabe 5 - Binäre Suche

In der ersten Vorlesung wurde bereits der Algorithmus der binären Suche vorgestellt. Die binäre Suche halbiert den zu durchsuchenden Bereich schrittweise. Begonnen wird mit der Betrachtung des gesamten Feldes. Von diesem wird der Wert des mittleren Elements bestimmt. Ist dieses der gesuchte Wert, so ist die Suche beendet. Andernfalls wird die Suche auf in der Hälfte fortgesetzt, in der der gesuchte Wert aufgrund der Sortierung des Feldes liegen müsste. Die Suche endet erfolglos, wenn der noch zu durchsuchende Abschnitt des Feldes keine Elemente mehr enthält. Setze diesen Algorithmus sowohl *iterativ* als auch *rekursiv* in den folgenden Methoden um:

- `boolean binarySearchIterative( int[] sorted, int candidate )`
- `boolean binarySearchRecursive( int[] sorted, int candidate )`

Gehe davon aus, dass dem Parameter `sorted` ein Feld übergeben wird, das aufsteigend sortierte Werte enthält.

Die Lösung setzt die Idee unmittelbar um.