AlgoDat Übung 08: Sortieren, Backtracking

Hinweis: Für diese Übung haben Sie 2 Wochen Zeit und sie zählt doppelt so stark wie einwöchige Übungen für das Gesamtergebnis.

Aufgabe 1 (30%): Sortieren 1

Implementieren Sie den Quicksort-Algorithmus in Java und halten Sie dabei die folgende Schnittstelle ein:

```
package at.fhooe.mc.algodat.ue8;

public interface Sorter<T> {
	public void sort(Comparable<T>[] a);
}
```

Erstellen Sie eine Klasse Quicksort, die diese Schnittstelle implementiert:

```
package at.fhooe.mc.algodat.ue8;

public class Quicksort<T> implements Sorter<T> {
	public void sort(Comparable<T>[] a) {
		qsort(a, 0, a.length-1);
	}

	public void qsort(Comparable<T>[] a, int from, int to) {
		// TODO
	}
}
```

Sie sehen, dass der Algorithmus auf beliebigen Objekten angewandt werden kann, die das Comparable-Interface implementieren. Beim Instantiieren für einen konkreten Typ (z.B. Integer) können Sie den generischen Typ T in der obigen Definition durch den Objekttyp ersetzen lassen, z.B.:

```
Sorter<Integer> qsort = new Quicksort<Integer>();
Integer[] a = new Integer[10];
for (int i=0; i<a.length; i++)
	a[i] = new Integer(9-i);
qsort.sort(a);
```

Testen Sie Ihren Algorithmus wie üblich mit JUnit-Tests und legen Sie besonders auf Grenzfälle wert (z.B. leeres Array, Array mit lauter gleichen Werten, umgekehrt sortiertes Array, etc.). Testen Sie Ihren Algorithmus auch mit mindestens 2 verschiedenen Typen, die das Comparable-Interface implementieren (z.B. neben Integer auch mit String).

Aufgabe 1 **(30%): Sortieren 2**

Implementieren Sie den Mergesort-Algorithmus in Java und halten Sie dabei die folgende Schnittstelle ein:

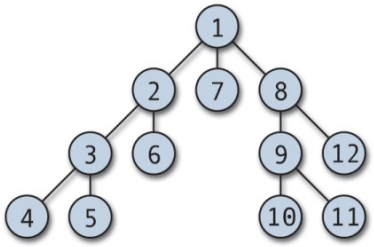
```
package at.fhooe.mc.algodat.ue8;

public class Mergesort<T> implements Sorter<T> {
	@Override
	public void sort(Comparable<T>[] a) {
		// TODO
	}
}
```

Testen Sie Ihren Mergesort-Algorithmus mit den gleichen Testfällen wie Ihren Quicksort-Algorithmus. Messen Sie außerdem die Geschwindigkeiten Ihrer Implementierungen für Integer-Arrays der Längen 100, 1.000 und 10.000 und für String-Arrays der Längen 100, 1.000 und 10.000 mit (unterschiedlichen) Strings der konstanten Länge von 100 Zeichen. Geben Sie die Laufzeiten in einer Tabelle zum Vergleich an.

Aufgabe 3 **(40%): Backtracking**

In Übung 3 haben Sie die Kernmethoden zur Lösung eines kombinatorischen Problems, nämlich die Prüfung, ob eine Lösung gültig ist und die die Ausgabe einer gültigen Lösung, implementiert. Sie werden allerdings gesehen haben, dass das Rahmenwerk zum Iterieren über alle Lösungskandidaten weder erweiterbar noch sehr elegant formuliert wurde.

Implementieren Sie daher einen vereinfachten Backtracking-Algorithmus in Java, der die verschachtelten Schleifen aus Aufgabe 1 in Übung 3 ablöst. Backtracking ist vergleichbar mit einer Tiefensuche in einem Baum, wobei die Blätter in diesem Baum die vollständigen Lösungen repräsentieren und die Nicht-Blatt-Knoten (also alle, welche noch Nachfolger haben), unvollständige Lösungskandidaten darstellen.



Wie bei einer Tiefensuche steigt der Backtracking-Algorithmus also bis zur Blattebene hinab, bevor ein Lösungskandidat auf Gültigkeit geprüft wird. In der allgemeinen Formulierung bricht der Algorithmus die Suche in einem Teilbaum der möglichen Lösungskandidaten bereits ab, wenn auch bei einem noch unvollständigen (partiellen) Lösungskandidaten schon klar ist, dass die Lösung insgesamt nicht mehr gültig werden kann. (Daher hat der oben gezeigte Suchbaum über alle Lösungen nicht alle Elemente in der Blattebene befüllt.)

Für diese Aufgabe ist es allerdings ausreichend, wenn Sie eine vereinfachte Variante von Backtracking implementieren und über **alle möglichen** Lösungskandidaten iterieren, um diese dann wie in Übung 3 zu prüfen. Um systematisch über alle Lösungskandidaten zu iterieren, schreiben Sie dazu zwei Methoden zur Ermittlung des ersten sowie des jeweils nächsten Lösungskandidaten aus dem aktuellen:

```
/** Set the first valid state for the visitors[] array */
public void setStartState() { … }

/** Based on the current state of the visitors[] array, set the
    next valid state.
 *
 * @param level determines the start position in the visitors[]
                array from which to change values.
 * @return true if the new state is valid, false if no new state
           could be found for this level.
 */
public boolean nextState(int level) { … }
```

Verwenden Sie nun eine vereinfachte Methode zum Evaluieren aller Lösungskandidaten:

```
public void printAllValid() {
  setStartState();

  do {
        // evaluate candidate
        if (isValid(visitors)) {
              /* print */
        }
  } while (nextState(0));
}
```

Hinweis: Die Methode `nextState` darf rekursiv oder iterativ formuliert sein. Sie können 5% Zusatzpunkte bekommen, wenn `nextState` iterativ formuliert wird und Sie dadurch die Methodensignatur auf folgende Variante ohne Parameter ändern können:

```
public boolean nextState() { … }
```