Übung 11: Sortieren

Abgabetermin: 14.06.2016

Name: _____ **Matrikelnummer:** _____

Gruppe: ☐ **G1** Di 10:15-11:00 ☐ **G2** Di 11:00-11:45 ☐ **G3** Di 12:45-13:30

Aufgabe	Punkte	gelöst	abzugeben schriftlich	abzugeben elektronisch	Korr.	Punkte
Aufgabe 1	6		Java-Programm und Testergebnisse	Java-Programm und Testergebnisse		
Aufgabe 2	10		Java-Programm und Testergebnisse	Java-Programm und Testergebnisse		
Aufgabe 3	8		Java-Programm und Testergebnisse	Java-Programm und Testergebnisse		

Aufgabe 1: HeapSort

Implementieren Sie eine einfache Variante des *HeapSort* mit folgender Schnittstelle:

```
class SimpleHeapSort extends Sorter {
  Object[] sort(Object[] items, Comparator cmp) { ... }
}
abstract class Sorter {
  abstract Object[] sort(Object[] items, Comparator cmp);
}
```

Die *sort*-Methode sortiert die Objekte im übergebenen Array in der Reihenfolge, die vom übergebenen *Comparator* vorgegeben wird. Fügen Sie dafür zunächst alle Elemente des Arrays in eine *ArrayPriorityQueue* ein (in der Vorgabedatei vorhanden). Befüllen Sie dann wiederum das Array neu, indem Sie Elemente einzeln aus der Queue holen und in das Array schreiben. Schließlich soll das ursprünglich übergebene und jetzt sortierte Array zurückgegeben werden.

Verwendungsbeispiel:

```
Sorter heapSort = new SimpleHeapSort();
String[] values = new String[] { "Maria", "Anton", "Valerie", "Birgit", "Claus" };
String[] sorted = heapSort.sort(values, String.CASE_INSENSITIVE_ORDER);
for (int i = 0; i < sorted.length; i++) {
  Out.print(" " + sorted[i]);
}  // Ausgabe: Anton Birgit Claus Maria Valerie
```

Abzugeben ist: Java-Programm und Testergebnisse

Aufgabe 2: MergeSort

Implementieren Sie das Sortierverfahren MergeSort:

```
class MergeSort extends Sorter {
  Object[] sort(Object[] items, Comparator cmp) {  ... }
}
```

Die *sort*-Methode wird verwendet wie in Aufgabe 1. Implementieren Sie zunächst eine Methode *mergeSorted(...)*, die zwei bereits sortierte Arrays in ein sortiertes Array zusammenfügt:

```
Object[] mergeSorted(Object[] a, Object[] b, Comparator cmp) { ... }
```

Implementieren Sie dann *sort(...)* so, dass es das übergebene Array in zwei Hälften teilt und auf jede Hälfte wiederum rekursiv *sort(...)* aufruft, wo dieses Array erneut halbiert und *sort(...)* auf die Hälften aufgerufen wird, und so weiter, bis das geteilte Array aus höchstens einem Element besteht. Bei der schrittweisen Rückkehr aus den rekursiven Aufrufen sollen die nun sortierten Teilarrays mit *mergeSorted(...)* zusammengefügt und zurückgegeben werden.

Abzugeben ist: Java-Programm und Testergebnisse

Aufgabe 3: Binäre Suche

Die binäre Suche ist ein einfacher, aber sehr effizienter Algorithmus zur Suche in sortierten Arrays. Zunächst wird das Element in der Mitte des Arrays bestimmt. Ist das gesuchte Element kleiner, wird die Suche in der Hälfte links von diesem Element fortgesetzt, sonst in der rechten Hälfte. Für dieses neue Suchintervall wird wiederum das mittlere Element bestimmt und mit dem gesuchten Element verglichen. Dieses Vorgehen wird so lange fortgesetzt, bis der gesuchte Wert gefunden wurde oder das neue Suchintervall leer ist.

Beispiel für die Suche nach *42* in folgendem Array:

(Hinweis: bei einer ungeraden Anzahl Elemente wird das kleinere mittige Element ausgewählt)

4	8	15	16	**23**	42	108	316	815	7418880

4	8	15	16	23	42	108	**316**	815	7418880

4	8	15	16	23	**42**	108	316	815	7418880

Der gesuchte Wert wird in diesem Beispiel in drei Schritten gefunden, während bei einer naiven Suche sechs Schritte notwendig wären. Allgemein sind nie mehr als $\lceil log_2(N) \rceil$ Schritte erforderlich (wobei N die Gesamtanzahl der Elemente ist).

Implementieren Sie die binäre Suche mit folgender Schnittstelle:

```
class BinarySearch extends Searcher {
  int search(Object[] haystack, Object needle, Comparator cmp) {  ... }
}
```

Die Methode *search* durchsucht das sortierte Object[]-Array *haystack* nach dem Objekt *needle* und verwendet den übergebenen *Comparator* zum Bewegen durch das Array. Zurückgegeben wird der Index des als erstes gefundenen übereinstimmenden Elements oder -1, wenn kein solches Element gefunden wurde. Verwendungsbeispiel:

```
Searcher binarySearch = new BinarySearch();
String[] values = new String[] { "Anton", "Birgit", "Claus", "Maria", "Valerie" };
int index = binarySearch.search(values, "Birgit", String.CASE_INSENSITIVE_ORDER);
Out.print(" Found Birgit at index " + index); // Ausgabe: 1
index = binarySearch.search(values, "Franz", String.CASE_INSENSITIVE_ORDER);
Out.print(" Found Franz at index " + index); // Ausgabe: -1
```

Sie können den Algorithmus sowohl iterativ als auch rekursiv mit einer Hilfsmethode implementieren.

Abzugeben ist: Java-Programm und Testergebnisse