Programmieren 1
Lernteam Semesterwoche 12

Aufgabe 1: Weitere Aufgaben aus dem Buch

1. Bearbeiten Sie die Aufgaben 8.2 und 8.5 zum DoME Beispiel
2. Diskutieren Sie in Gruppen die Aufgaben 8.9 und 8.10.
3. Diskutieren Sie die Aufgabe 8.13.

Aufgabe 2: Quicksort

1. Implementieren Sie in der vorgegebenen Klasse QuickSort (siehe ILIAS) eine main(…) Methode mit Testfällen.

```
    public static void main(String[] args)
    {
        int[] i = {4,13,50,1,2,33,8,123};
        quickSort(i, 0, 7);
        print(i);
    }
```

```
C:\temp\BlueJ Projects\Sort>java Quicksort
1
2
4
8
13
33
50
123
```

2. Implementieren Sie eine zweite Version von Quicksort, mit der ‚median-of-three' Verbesserung.
http://www.people.carleton.edu/~grossm/

```
//Lösungsbeispiel Quicksort Algorithmus mit „Median-of-three" Optimierung
/* Copyright 2008 - Hochschule Luzern */
/**
 * Eine Implementationenen von Quicksort.
 *
 * @author Peter Sollberger
 * @version 1.2, 31. Oktober 2008
 */
public class Quicksort
{

    public static void main(String[] args)
    {
        test();
    }
    /**
     * Konstruktor macht nichts.
     */
    public Quicksort()
    {
        test();
    }

    private static void test()
    {
        int[] i = {8,2,4,9,5,3,5,7,6,10,9,1,13};
        quickSortM(i, 0, i.length-1);
```

```
    }

    /**
     * Klassischer Quicksort Algorithmus mit rechtem Trennelement.
     * @param a Array zum Sortieren
     * @param left linke Grenze
     * @param right rechte Grenze
     */
    public static void quickSort(int a[], int left, int right)
    {
        int up   = left;                // linke Grenze
        int down = right - 1;           // rechte Grenze (ohne Trennelement)
        int t    = a[right];            // rechtes Element als Trennelemt
        boolean allElementChecked = false;

        do {
            while (a[up] < t) {
                up++;                   // suchen größeres Element von links an
            }
            while ((a[down] > t) && (down > up)) {
                down--;                 // suchen kleineres Element von rechts an
            }
            if (up < down) {
                exchange(a, up, down);  // austauschen
                up++;                   // linke und rechte Grenze verschieben:
                down--;
            }
            else
            {
                allElementChecked = true;
            }
        } while (!allElementChecked);   // Überschneidung

        exchange(a, up, right);         // Trennelement an endgültige Position
(a[up])

        if (left < up - 1) {            // mehr als 1 Element in linker Teilfolge?
            quickSort(a, left, up - 1); // linke Hälfte sortieren
        }
        if (up + 1 < right) {           // mehr als 1 Element in rechter
Teilfolge?
            quickSort(a, up + 1, right); // rechte Hälfte sortieren (ohne
Trennelement)
        }
    }

    /**
     * Quicksort Algorithmus mit mittlerem Trennelement (Median of three).
     * @param a Array zum Sortieren
     * @param left linke Grenze
     * @param right rechte Grenze
     */
    public static void quickSortM(int a[], int left, int right)
    {
        int up   = left;                // linke Grenze
        int down = right - 1;           // rechte Grenze (ohne Trennelement)
        //Index des Trennelements bestimmen (Median of three)
        int tindex = getMedianIndex(a, left, (left + (right - left) / 2), right);
        int t    = a[tindex];            // Mittelwertiges Element als Trennelemt
        exchange(a, tindex, right);     // Trennelement an rechte Grenze
verschieben

        boolean allElementChecked = false;

        do {
            while (a[up] < t) {
                up++;                   // suchen größeres Element von links an
            }
            while ((a[down] > t) && (down > up)) {
                down--;                 // suchen kleineres Element von rechts an
            }
```

```
            if (up < down) {
                exchange(a, up, down);   // austauschen
                up++;                    // linke und rechte Grenze verschieben:
                down--;
            }
            else
            {
                allElementChecked = true;
            }
        } while (!allElementChecked);    // Überschneidung

        exchange(a, up, right);          // Trennelement an endgültige Position
(a[up])

        if (left < up - 1) {             // mehr als 1 Element in linker Teilfolge?
            quickSortM(a, left, up - 1);  // linke Hälfte sortieren
        }
        if (up + 1 < right) {            // mehr als 1 Element in rechter
Teilfolge?
            quickSortM(a, up + 1, right); // rechte Hälfte sortieren (ohne
Trennelement)
        }
    }

    /*
     * Methode zur Bestimmung des Trennelements
     * @param value1 linke Grenze des Arrays
     * @param value2 mittleres Element des Arrays
     * @param value3 rechtes Element des Arrays
     */
    private static int getMedianIndex(int[] a, int index1, int index2, int index3)
    {
        int value1 = a[index1], value2 = a[index2], value3 = a[index3];
        int returnvalue = 0;
        int returnindex = 0;

        if((value1>=value2 && value1<=value3) || (value1>=value3 &&
value1<=value2))
        {
            returnindex = index1;
            returnvalue = value1;
        }
        else if(value2>=value1 && value2<=value3 || value2>=value3 &&
value2<=value1)
        {
            returnindex = index2;
            returnvalue = value2;
        }
        else if(value3>=value1 && value3<=value2 || value3>=value2 &&
value3<=value1)
        {
            returnindex = index3;
            returnvalue = value3;
        }

        System.out.println("Trennelement "+returnvalue+" bei Index "+returnindex);
        return returnindex;
    }

    /**
     * Vertausche im übergebenen Array die beiden Elemente an der Position
     * firstIndex und secondIndex.
     */
    private static void exchange(int a[], int firstIndex, int secondIndex)
    {
        System.out.println(""+a[firstIndex]+" mit "+a[secondIndex]+" austauschen");
        print(a);
        int tmp;

        tmp            = a[firstIndex];
        a[firstIndex]  = a[secondIndex];
```

```
        a[secondIndex] = tmp;
    }

    private static void print(int[] a)
    {
        for(int x=0; x<a.length; x++)
        {
            System.out.print(a[x]+" | ");

        }
        System.out.println();
    }
}
```

3. Vergleichen Sie die Rekursionstiefen der beiden Implementationen mit verschiedenen Testdaten (z.B. einer bereits sortierten Folge).

Aufgabe 3: Quicksort Speicherbedarf

Die Methode Quicksort ruft sich rekursiv auf. Bei jedem Aufruf werden die Übergabeparameter auf dem Stack abgelegt, und jeder Methodenaufruf belegt für seine Variablen ebenfalls einen gewissen Platz auf dem Stack. Nennen wir diesen Platzbedarf eine Quick-Speichereinheit.
1. Aus wie vielen Bytes besteht eine Quick-Speichereinheit beim klassischen Quicksort (Implementation gemäss Folien)?
2. Wie verändert sich die Quick-Speichereinheit beim ‚Quick-Insertion-Sort' Verfahren?
3. Wie gross (in Quick-Speichereinheiten) muss der Stack bei Quicksort im günstigsten und im ungünstigsten Fall sein?

Aufgabe 4: Mergesort (optional)

1. Implementieren Sie den Mergesort Algorithmus.
2. Vergleichen Sie die Geschwindigkeit von Mergesort mit dem klassischen Quicksort für verschiedenste Eingabedaten. Wählen Sie grosse, zufällige Folgen, damit Sie Geschwindigkeitsunterschiede messen können. Verwenden Sie dabei die Methode System.currentTimeMillis() zur Zeitmessung. Können Sie die Aussage aus der Vorlesung (Geschwindigkeitsvorteil von Quicksort: 10% - 50%) beobachten?

```
//Lösungsbeispiel                NOCH NICHT FERTIG!!
import java.util.*;

public class Mergesort
{
    private ArrayList a;
    public Mergesort()
    {
        a = new ArrayList();
        a.add(6);
        a.add(5);
        a.add(4);
        a.add(3);
        a.add(2);
        a.add(1);
        a.add(0);

        mergesort(0, (a.size()-1)/2, (a.size()-1));
        print(a);
    }

    public ArrayList mergesort(int si, int m, int ei)
    {
        if(a.size()-1<=1)
        {
            return a;
        }
        else
        {
            merge(mergesort(0, (a.size()-1) / 4, (a.size()-1)/2),
mergesort(a.size(), a.size()+(a.size()-1) / 2, (a.size()-1))) ;
        }
        return a;
    }

    /*
     * @param si Start Index
     * @param m Trennelement
     * @param ei End Index
     */
    public ArrayList merge(int si, int m, int ei)
    {

        ArrayList mergedArray = new ArrayList();


        int a1si = 0;                 //start index a1
        int a1ei = m;                 //end index a1
        int a2si = m+1;               //start index a2
        int a2ei = ei;                //end index a1

        while(a1si<=a1ei && a2si<=a2ei)
        {
           if((Integer)a.get(a1si) <= (Integer)a.get(a2si))
           {
               //falls das Element in Teilarray a1 kleiner ist als das Element von
Teilarray a2
               int debug = (Integer)a.get(a1si);
               mergedArray.add(a.get(a1si));
               a1si++;
```

```
            }
            else
            {
                int debug = (Integer)a.get(a2si);
                mergedArray.add(a.get(a2si));
                a2si++;
            }
        }

        while(a1si <= a1ei)
        {
            mergedArray.add(a.get(a1si));
            a1si++;
        }

        while(a2si <= a2ei)
        {
            mergedArray.add(a.get(a2si));
            a2si++;
        }



        return mergedArray;
    }

    private void print(ArrayList a)
    {
        for(int x=0; x<a.size()-1; x++)
        {
            System.out.println(a.get(x));
        }
    }
}
```