

Prof. Dr. C. Sohler, D. Köppl  
R. Kalkreuth, D. Siedhoff, I. Kamehkhosh, M. Ludewig  
<http://tiny.cc/dap2praktikum>

Sommersemester 2016

## DAP2 Praktikum – Blatt 5

Ausgabe: 11. Mai — Abgabe: 17. Mai–20. Mai

**Wichtig:** Der Quellcode ist natürlich mit sinnvollen Kommentaren zu versehen. Überlegen Sie außerdem in welchen Bereichen Invarianten gelten müssen und überprüfen Sie diese ggf. an sinnvollen Stellen mit *Assertions* (siehe Hinweis auf Blatt 2).

### Kurzaufgabe 5.1

(4 Punkte)

*Lernziel: Wiederholung Teile & Herrsche und Sortieralgorithmen: Quicksort*

Zur Wiederholung der Teile & Herrsche Algorithmen betrachten wir nun den Sortier-Algorithmus Quicksort. Schreiben Sie eine Klasse `Quicksort`, die analog zur Klasse `Sortierung` vom Praktikumsblatt 2 als Eingabewert einen Integerwert erwartet, der für die Anzahl der zu sortierenden Werte steht. Gehen Sie danach wie folgt vor:

- Erzeugen Sie ein Integerarray in der Größe des Eingabeparameters und befüllen Sie es mit zufälligen Werten.
- Implementieren Sie den Quicksort-Algorithmus (siehe nächste Seite) und lassen Sie ihn für das Array ausführen.
- Messen Sie die Laufzeit und vergleichen Sie die Werte mit Ihren Werten vom MergeSort-, InsertionSort- und BubbleSort-Algorithmus.
- Benutzen Sie die bereits von Ihnen erstellte Methode `isSorted`, um das Array auf Sortiertheit zu prüfen.

```

1: function QUICKSORT( $A, l, r$ )
2:   if  $l < r$  then
3:      $i \leftarrow l$ 
4:      $j \leftarrow r$ 
5:      $pivot \leftarrow A[(l + r)/2]$ 
6:     while  $i \leq j$  do
7:       while  $A[i] < pivot$  do
8:          $i \leftarrow i + 1$ 
9:       while  $A[j] > pivot$  do
10:         $j \leftarrow j - 1$ 
11:      if  $i \leq j$  then
12:         $tmp \leftarrow A[i]$ 
13:         $A[i] \leftarrow A[j]$ 
14:         $A[j] \leftarrow tmp$ 
15:         $i \leftarrow i + 1$ 
16:         $j \leftarrow j - 1$ 
17:      QUICKSORT( $A, l, j$ )
18:      QUICKSORT( $A, i, r$ )

```

### Kurzaufgabe 5.2

(4 Punkte)

*Lernziel: Gieriger Algorithmus Münzwechselproblem*

Wir betrachten nun das *Münzwechselproblem*: Wenn ein Fahrkartenautomat Wechselgeld ausgibt, so möchte man den Betrag sicherlich nicht in lauter 1-Cent-Münzen ausgezahlt bekommen. Im Normalfall möchte man den Betrag mit möglichst wenigen Münzen bzw. Scheinen erhalten. Der Automat hat also ein Optimierungsproblem zu lösen. Der Automat kennt die Wertigkeiten  $w_k, \dots, w_1$  der Währung, wobei  $w_k > w_{k-1} > \dots > w_1$  und  $w_1 = 1$  ist, und den Betrag  $B$ , der zurückgezahlt werden muss. Der Automat geht wie folgt vor: Ist  $B \geq w_k$ , dann gibt er  $n$  Münzen der Wertigkeit  $w_k$  aus mit  $0 \leq B - n \cdot w_k < w_k$  und verringert den Betrag entsprechend zu  $B \leftarrow B - n \cdot w_k$ . Danach führt er das gleiche für  $w_{k-1}$  aus und fährt anschließend auf gleiche Weise fort bis schließlich  $w_1$  erreicht ist. Implementieren Sie diese umgangssprachliche Beschreibung wie folgt:

- Schreiben Sie eine `main`-Methode, die als Eingabeparameter zwei Werte erwartet. Der erste Parameter soll ein String sein, der für die Währung steht, die benutzt werden soll. Dabei werden zwei unterschiedliche Strings akzeptiert: "Euro" und "Alternative". Bei Euro soll die Währung in der Form `int[] w = {200, 100, 50, 20, 10, 5, 2, 1}` angelegt werden, was den Werten der Münzstücken in Cent entspricht. Bei Alternative soll die Währung auf `int[] w = {200, 100, 50, 20, 10, 5, 4, 2, 1}` angelegt werden. Der zweite Parameter `b` soll die Höhe des Wechselgeldes angeben.
- Schreiben Sie die Methode `public static int[] change(b, int[] w)`, die den Münzwechselalgorithmus ausführt und das Ergebnisarray mit den zu den Werten der Münzen passenden Anzahlen zurückgibt.  
**Beispiel:** Für `b=455` und "Euro", soll das Ergebnisarray `{2, 0, 1, 0, 0, 1, 0, 0}` sein.
- Lassen Sie die `main`-Methode die Ergebnisse auf den Bildschirm ausgeben.

Liefert das implementierte Verfahren stets optimale Lösungen mit einer minimalen Anzahl an Münzen?