Übung 1 zu KMPS Lösung

Aufgabe 1:
Implementieren Sie den Sortierungsalgorithmus *Quicksort* iterativ in Java.

Lösung:
Zur Auflösung der Rekursion benötigt man üblicherweise einen Stack (Ausnahme sind sogenannte primitiv-rekursive Funktionen, wie z.B. Fibonacci. Da kommt man ohne aus.). Das geht bei Quicksort nicht so einfach. Darum folgende Lösung mit Stack:

Dabei wird mit der Hilfsfunktion Partition das Feld so umsortiert, dass kleinere Elemente nach links und größere nach rechts kommen und die Trennposition zurückgegeben wird. Die beiden Teile müssen dann weiterbearbeitet werden (was normalerweise rekursiv passiert). Das geht aber hier iterativ, indem man jeweils die Grenzen der beiden Teile auf den Stack legt und diese dann analog weiterbearbeitet, bis der Stack leer ist.

Falls Sie dieses mit den beiden rekursiven Implementierungen in Foliensatz V02 vergleichen, sieht man sehr schön, wieviel einfacher das Ganze mit Rekursion geht und dann nochmals einfacher, wenn man Higher-Order Funktionen verwendet.

```
//Quelle: http://www.techiedelight.com/iterative-implementation-of-quicksort/

import java.util.Arrays;
import java.util.Stack;

// Simple pair class in Java
class Pair {
    private final int x;
    private final int y;

    Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }
}

class QuickSort
{
    public static void swap (int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    public static int Partition(int a[], int start, int end)
    {
        // Pick rightmost element as pivot from the array
        int pivot = a[end];

        // elements less than pivot will go to the left of pIndex
        // elements more than pivot will go to the right of pIndex
        // equal elements can go either way
        int pIndex = start;

        // each time we finds an element less than or equal to pivot,
        // pIndex is incremented and that element would be placed
        // before the pivot.
        for (int i = start; i < end; i++)
        {
            if (a[i] <= pivot)
            {
                swap(a, i, pIndex);
```

Übung 1 zu KMPS Lösung

```
                pIndex++;
            }
        }
        // swap pIndex with Pivot
        swap (a, pIndex, end);

        // return pIndex (index of pivot element)
        return pIndex;
    }

    // Iterative Quicksort routine
    public static void iterativeQuickSort(int[] a)
    {
        // stack of std::pairs for storing subarray start and end index
        Stack<Pair> stack = new Stack<>();

        // get starting and ending index of given array (vector)
        int start = 0;
        int end = a.length - 1;

        // push array start and end index to the stack
        stack.push(new Pair(start, end));

        // run till stack is empty
        while (!stack.empty())
        {
            // pop top pair from the list and get sub-array starting
            // and ending indices
            start = stack.peek().getX();
            end = stack.peek().getY();
            stack.pop();

            // rearrange the elements across pivot
            int pivot = Partition(a, start, end);

            // push sub-array indices containing elements that are
            // less than current pivot to stack
            if (pivot - 1 > start) {
                stack.push(new Pair(start, pivot - 1));
            }

            // push sub-array indices containing elements that are
            // more than current pivot to stack
            if (pivot + 1 < end) {
                stack.push(new Pair(pivot + 1, end));
            }
        }
    }

    // Iterative Implementation of Quicksort
    public static void main(String[] args)
    {
        int a[] = { 9, -3, 5, 2, 6, 8, -6, 1, 3 };

        iterativeQuickSort(a);

        // print the sorted array
        System.out.println(Arrays.toString(a));
    }
}
```

Übung 1 zu KMPS Lösung

Aufgabe 2:
Wofür kann der Datentyp *Unit* in Scala verwendet werden? Gibt es Parallelen zum Type-System in Java?

Lösung:
Prinzipiell als „void"-Ersatz.

Detaillierte Antwort und Unterschiede:
http://james-iry.blogspot.com/2009/07/void-vs-unit.html

Aufgabe 3:
a) Stellen Sie die folgende Liste in Prolog-Notation dar ohne Verwendung von „,": `1::2::3::Nil`
b) Sind die Terme links und rechts von `::` rechts- oder linksassoziativ zu klammern?

Lösung:
a) `[1|[2|[3|[]]]]`
b) rechtsassoziativ, da rechts vom Zeichen eine Liste steht und links nur ein Element.

Aufgabe 4:
Geben Sie eine Java-Methode an, die bei gleicher Eingabe verschiedene Lösungen liefert.

Lösung:
```
class Seiteneffekt {
  boolean mutable = true;
  boolean change() {mutable = !mutable; return mutable}
}
```

Aufgabe 5:
In Prolog gibt es keine verschachtelten Relationen.
Wie muss man in Prolog vorgehen, um verschachtelte Funktionsaufrufe, wie `square(square(4))` auszuwerten.

Lösung:
Man muss den verschachtelten Aufruf entschachteln unter Ausnutzung des zusätzlichen Arguments.
Für obiges Beispiel ergibt sich: ?- `square(4,H),square(H,Z).`

Aufgabe 6:
Implementieren Sie eine Methode bzw. Funktion zur Berechnung der Fibonaccizahlen

a) iterativ in Java
b) rekursiv in Scala

Kann b) in einer Zeile (ohne Semikolon) programmiert werden?

Übung 1 zu KMPS Lösung

Wie groß ist jeweils der Zeit- und Speicheraufwand?

Lösung:
a)

```
public int fibIterative(int n) {
    int n1 = 0, n2 = 1, n3;
    if (n==0 || n==1) return n;
    for (int i = 2; i <= n; i++) {
        n3 = n1 + n2;
        n1 = n2;
        n2 = n3;
    }
    return n2;
}
```

b)

```
def fib(n: Int) : Int = if (n==0 || n==1) n else fib(n-1) + fib(n-2)
```

In a) Zeit-Aufwand linear und Speicheraufwand konstant
In b) beide Aufwände exponentiell.

Aufgabe 7:
Die untenstehende Methode `ggTIterativ (int, int)` implementiert den euklidischen Algorithmus zur Berechnung des größten gemeinsamen Teilers iterativ in Java.
Implementieren Sie hiervon ausgehend den *ggT* rekursiv in Scala.

Wie groß sind jeweils der Zeit- und Speicheraufwand beim iterativen bzw. rekursivem Algorithmus?

```
public int ggTIterativ(int a, int b) {
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

Lösung:

```
def ggT(a: Int, b: Int): Int = if (b == 0) a else ggT(b, a % b)
```

Laufzeitaufwand ist jeweils linear. Speicheraufwand ist konstant, falls die tail-Rekursion der Funktion zur Optimierung genutzt wird, linear, falls nicht.
