VL05, Lösung 1

a) Die erste Methode gibt eine gegebene Zahl vom Typ `int` rückwärts auf der Standardausgabe wieder aus: 4321 und 8765.

Sei z die Anzahl der Ziffern von n. Da eine Zahl mit Wert n im Dezimalsystem $z = \lfloor \log_{10} n \rfloor + 1$ Ziffern zur Darstellung benötigt, hat die Methode eine Laufzeit von $O(z) = O(\log n)$.

b) Auch die zweite Methode spiegelt die gegebene Zahl vom Typ `int`, gibt diese jedoch nicht aus, sondern liefert sie als Rückgabewert an den Aufrufer zurück:

`n`	`logn`	`zehnHochLogn`	Rückgabewert
5678	3	1000	8000 +
567	2	100	700 +
56	1	10	60 +
5	0	1	5

Aufgrund der Endrekursion hat diese Methode dieselbe Laufzeit $O(z) = O(\log n)$, wenn wir zusätzlich voraussetzen, dass die Java-Methoden `Math.log10` und `Math.pow` konstante Laufzeit haben.

c) Iterative Implementierung von `rev1` mit Ausgabe der Ziffern:

```
public static void rev1iter(int n)
{
    assert(n >= 0);

    do                // 0 muss auch ausgegeben werden
    {
        System.out.print(n % 10);
        n /= 10;
    }
    while (n > 0);
}
```

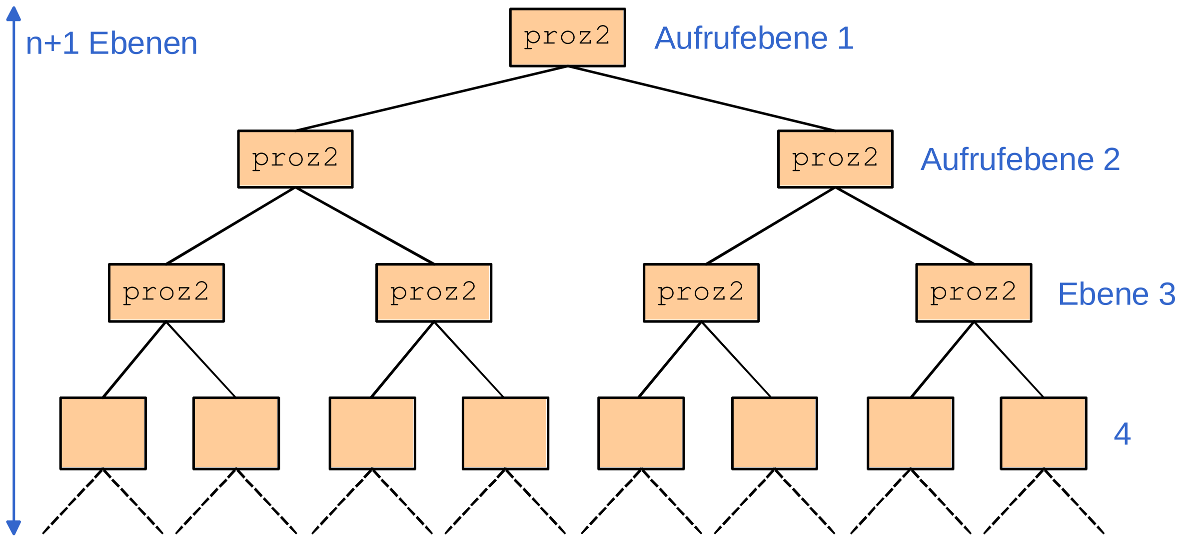
Iterative Implementierung von `rev2` mit Rückgabe der gespiegelten Zahl:

```
public static int rev2iter(int n)
{
    assert(n >= 0);

    int ergebnis = 0;

    while (n > 0)
    {
        ergebnis = 10 * ergebnis + n % 10;
        n /= 10;
    }

    return ergebnis;
}
```

VL05, Lösung 2

Methode	Aufrufe für $n=3$	Als Funktion von n	Asymptotische Komplexität
`funk1`	3	n	$O(n)$
`proz2`	1+2+4+8 = 15	$2^{n+1}-1$	$O(2^n)$

Da `proz2` jeweils zwei weitere Aufrufe von `proz2` startet falls $n > 0$, ergibt sich folgende Aufrufstruktur (analog zu den Türmen von Hanoi oder einer Kernspaltung):



In Ebene 1 erfolgt ein Aufruf, in Ebene 2 erfolgen zwei Aufrufe, in Ebene 3 vier Aufrufe, in Ebene 4 acht Aufrufe, und so weiter. In Ebene n+1 erfolgen somit 2^n Aufrufe. Insgesamt sind es 1 + 2 + 4 + 8 + … + 2^n Aufrufe. Es gilt:

$$\sum_{i=1}^{n+1} 2^{i-1} = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Es gibt somit insgesamt $2^{n+1}-1$ Aufrufe von `tuwas`, da jeder Aufruf von `proz2` einen Aufruf von `tuwas` enthält.

VL05, Lösung 3

a) $\mathrm{Ulam}(2) = \mathrm{Ulam}(1) = 1$
$\mathrm{Ulam}(3) = \mathrm{Ulam}(10) = \mathrm{Ulam}(5) = \mathrm{Ulam}(16) = \mathrm{Ulam}(8) = \mathrm{Ulam}(4) = \mathrm{Ulam}(2) = 1$

b)
```
public static int UlamRekursiv(int n)
{
    return (n <= 1) ? 1 : UlamRekursiv(n % 2 == 0 ? n / 2 : 3 * n + 1);
}

public static int UlamIterativ(int n)
{
    while (n > 1)
        n = (n % 2) == 0 ? n / 2 : 3 * n + 1;

    return n;
}
```

VL05, Lösung 4

```
public class Fibonacci
{
    public static long fibRekursiv(final int n)
    {
        assert(n >= 0);

        return (n <= 1) ? n : fibRekursiv(n - 1) + fibRekursiv(n - 2);
    }

    public static long fibIterativ(int n)
    {
        if (n == 0)
            return 0;

        long fibVorletzter = 0;
        long fibLetzter = 1;

        while (n-- >= 2)
            fibLetzter = fibVorletzter + (fibVorletzter = fibLetzter);

        return fibLetzter;
    }
}
```

Die Lösung zu b) verwendet zur Zeitmessung die Klasse `StopUhr` aus Übung 2.

```
public class FibonacciTest
{
    public static void main(String[] args)
    {
        StopUhr meineUhr = new StopUhr();

        meineUhr.start();
        for (int a = 1; a <= 50; a++)
            Fibonacci.fibRekursiv(a);
        meineUhr.stop();

        System.out.println("Laufzeit bei rekursiver Berechnung: " +
             meineUhr.getDuration()/1000000.0 + " msec");

        meineUhr.start();
        for (int a = 1; a <= 50; a++)
            Fibonacci.fibIterativ(a);
        meineUhr.stop();

        System.out.println("Laufzeit bei iterativer Berechnung: " +
             meineUhr.getDuration()/1000000.0 + " msec");
    }
}
```

Das Laufzeitverhalten der iterativen Methode ist $O(n)$, die rekursive Methode hat eine Laufzeit von $O(2^n)$. Die exponentielle Laufzeit entsteht, weil auf jeder Rekursionsstufe zwei neue Rekursionen erzeugt werden, und so immer wieder dieselben Werte errechnet werden.

VL05, Lösung 5

Das Laufzeitverhalten dieser Implementierung ist $O(n)$, trotz Rekursion. Die Methode bildet das iterative Verfahren zur Berechnung rekursiv ab, das heißt auf jeder Rekursionsstufe wird höchstens ein weiterer Aufruf als Endrekursion erzeugt.