

Vorname	Nachname	Matrikel-Nr
---------	----------	-------------

**Aufgabe 1** (15 Punkte): Betrachten Sie die folgende (rekursive!) Funktion:

```

1 static int reku(int n) {
2     if (n < 0) {
3         return reku(-n);
4     } if (n > 20) {
5         return n;
6     } else if(n % 2 == 0) {
7         return reku(n*3 + 1);
8     } else if (n % 3 == 0) {
9         return n / 3;
10    } else {
11        return reku(n*5 + 1);
12    } // if
13 } // reku

```

Tragen Sie in die folgende Wertetabelle die fehlenden **Funktionswerte** ein. Ermitteln Sie diese Funktionswerte, indem Sie die Funktion **reku** mehrmals "mit Papier und Bleistift" **ausführen**.

n	-1	0	1	2	3	4
reku(n)						

**Aufgabe 2** (20 Punkte): Schreiben Sie eine **rekursive** Funktion namens **rekuSumm** entsprechend der folgenden Beschreibung:

```

1 static int rekuSumm(int[] ir, int ind);
2 // Liefert die Summe der Komponenten der Reihung ir, deren Index
3 // groesser oder gleich ind ist (d.h. "die Summer aller Komponenten
4 // ab der Stelle ind"). Beispiel:
5 // int[] ir01 = {5, 2, 4};
6 // rekuSumm(ir01, 0) ist gleich 11 (weil 5 + 2 + 4 gleich 11 ist)
7 // rekuSumm(ir01, 1) ist gleich 6 (weil 2 + 4 gleich 6 ist)
8 // rekuSumm(ir01, 2) ist gleich 4 (weil 4 gleich 4 ist)
9 // rekuSumm(ir01, 3) ist gleich 0 (weil kein Index von ir01 groesser
10 // oder gleich 3 ist)

```

Ihre Lösung darf **keine Schleife** enthalten, sondern muss "rein **rekursiv**" funktionieren.

**Aufgabe 3** (10 Punkte): Betrachten Sie die folgende **Grammatik für Ganzzahl-Literale**:

```

R01: Lit -> Ziff      R07: Ziff -> '4'
R02: Lit -> Ziff Lit   R08: Ziff -> '5'
R03: Ziff -> '0'       R09: Ziff -> '6'
R04: Ziff -> '1'       R10: Ziff -> '7'
R05: Ziff -> '2'       R11: Ziff -> '8'
R06: Ziff -> '3'       R12: Ziff -> '9'

```

Dabei sind Lit und Ziff **Zwischensymbole**, Lit ist das **Startsymbol** und '0', '1', ..., '9' sind **Endsymbole**. Leiten Sie (aus dem Startsymbol Lit) das Wort **213** ab.

**Achtung:** Als Lösung dieser Aufgabe sollen Sie eine **Ableitung** angeben (eine Folge von Zeilen mit einer Regel-Nr. wie R01 oder R11 etc. zwischen je zwei Zeilen) und **nicht** einen **Syntaxbaum**.

**Aufgabe 4** (20 Punkte): In Java sind Ganzzahl-Literale wie z.B. **100000000** oder **1000000000** schwer zu lesen und zu unterscheiden. In der Programmiersprache **Ada** darf man in einem Ganzzahl-Literal **zwischen je zwei Ziffern** einen **Unterstrich** einfügen (um damit das Literal **lesbarer** zu gestalten). Ein Unterstrich darf aber weder als **erstes** Zeichen noch als **letztes** Zeichen eines Literals geschrieben werden und **zwei** Unterstriche nebeneinander sind auch **nicht** erlaubt.

**Beispiele** für Ganzzahl-Literale, die in Ada **erlaubt** sind:

```

1 12345           // Null Unterstriche, wie in Java
2 12_345          // Ein Unterstrich, gut lesbar!
3 1_23_45         // Zwei Unterstriche, gut lesbar?
4 1_2_3_4_5       // Vier Unterstriche, gut lesbar??
5 1_000_000_000   // Drei Unterstriche, eine Milliarde.
6 100_000_000     // Zwei Unterstriche, 100 Millionen.

```

**Beispiele** für Ganzzahl-Literale, die in Ada **nicht erlaubt** sind:

```

1 _123           // Ein Unterstrich als erstes Zeichen
2 123_          // Ein Unterstrich als letztes Zeichen
3 1__2          // Zwei Unterstriche nebeneinander

```

Geben Sie eine (kontextfreie, Typ2-) **Grammatik** für die Sprache aller solcher Ganzzahl-Literale an. Verwenden Sie dabei möglichst nur die beiden Zwischensymbole **Lit** und **Ziff** und die elf Endsymbole **'\_'**, **'0'**, **'1'**, **'2'**, ..., **'9'**.

**Aufgabe 5** (15 Punkte): In jeder Teilaufgabe dieser Aufgabe sollen Sie zwei Formen von Sammlungen miteinander vergleichen (z.B. **Sortierte Reihungen** und **Unsortierte Reihungen**) und einen **Vorteil** angeben, den die **erste** Form im Vergleich zur **zweiten** Form hat. Falls es mehrere solche Vorteile gibt, sollen Sie **einen** (möglichst wichtigen) davon auswählen und angeben. Beschreiben Sie den Vorteil mit einem möglichst **kurzen** Satz.

Nr. der Teilaufgabe	Erste Sammlungsform	Zweite Sammlungsform
1	Sortierte Reihungen	Unsortierte Reihungen
2	Unsortierte Reihungen	Sortierte Reihungen
3	Verkettete Listen	Reihungen
4	Sortierte Reihungen	Sortierte Verkettete Listen
5	Bäume	Hash-Tabellen
6	Hash-Tabellen	Bäume

**Aufgabe 6** (10 Punkte):

- Welche **Tiefe** muss ein binärer Baum mindestens haben, damit er 2000 Knoten enthalten kann?
- Wieviele Knoten** kann ein binärer Baum der Tiefe 7 höchstens enthalten?
- Vereinbaren Sie eine **Class**-Variable **kob** und eine **String**-Variable **str**. Weisen Sie der Variablen **kob** auf 3 verschiedene Weisen das **Class**-Objekt zu, welches die Klasse **String** repräsentiert (oder: reflektiert).
- Geben Sie **vier Bruchzahlen** an, 1. eine große und genaue, 2. eine kleine und genaue, 3. eine große und ungenaue und 4. eine kleine und ungenaue.

**Lösung 1** (15 Punkte):

n	-1	0	1	2	3	4
reku(n)	96	96	96	36	1	66

**Lösung 2** (20 Punkte):

```

1  static int rekuSumm(int[] ir, int ind) {
2      // Liefert die Summe der Komponenten der Reihung ir, deren Index
3      // groesser oder gleich ind ist.
4
5      if (ind > ir.length-1) return 0;
6      return ir[ind] + rekuSumm(ir, ind+1);
7  } // rekuSumm

```

**Lösung 3** (10 Punkte): Eine Ableitung des Wortes 213:

```

      Lit
R02   Ziff Lit
R02   Ziff Ziff Lit
R01   Ziff Ziff Ziff
R05   Ziff Ziff Ziff
      2   Ziff Ziff
R04   2   1   Ziff
R06   2   1   3

```

**Lösung 4** (20 Punkte): Eine Grammatik für Ganzzahl-Literale, in denen zwischen zwei Ziffern ein Unterstrich stehen darf:

```

R01: Lit -> Ziff
R02: Lit -> Ziff Lit
R03: Lit -> Ziff '_' Lit
R04: Ziff -> '0'
R05: Ziff -> '1'
...
R13: Ziff -> '9'

```

**Lösung 5** (15 Punkte):

- 5.1. Das **Suchen im negativen Fall** ist (bei sortierten R. im Vergleich zu unsortierten R.) **schneller**.
- 5.2. Das **Einfügen** ist (bei unsort. R. im Vergleich zu sort. R.) **schneller**. Ebenso das **Entfernen**.
- 5.3. Eine verkettete Liste ist **erweiterbar**, flexibel, eine Reihung hat eine **feste**, starre **Länge**.
- 5.4. Das **Suchen** ist (in sort. Reihungen im Vergleich zu sort. verketteten Listen) **schneller**.
- 5.5. Die Komponenten eines Baumes kann man leicht **sortiert** (nach ihren Schlüsseln) **ausgeben** oder bearbeiten, bei einer Hash-Tabelle ist das sehr schwer.
- 5.6. Das **Suchen** ist (bei Hash-Tabellen im Vergleich zu Bäumen) **schneller** und unabhängig von der Größe der Sammlung.

**Lösung 6** (10 Punkte):

6.1. **Die Tiefe 11!**

6.2. **Höchstens 127 Knoten!**

6.3. kob=Class.forName("java.lang.String"); kob=String.class; kob=str.getClass();

6.4. 1.) 1.111111111111111111111111E50, 2.) 2.222222222222222222E-50, 3.) 3.3E50, 4. ) 4.4E-50.