3. Übung

Abgabe bis 22.05.2017, 10:00 Uhr

Einzelaufgabe 3.1: Vollständige Induktion **13 EP**

Gegeben Sei folgende Methode `f`:

```
long f(int a, int u, int d) {
    if (d == 1) {
        return a;
    } else if (d == 2) {
        return 2 * a + u;
    } else {
        return f(a, u, d - 2) + 2 * a + 2 * d * u - 3 * u;
    }
}
```

Beweisen Sie *formal* mittels *vollständiger Induktion*: $\forall d \geq 1$: `f(a,u,d)` $\equiv \dfrac{d(2a+(d-1)u)}{2}$

Sie dürfen dabei vereinfachend annehmen, dass es keinen Überlauf geben kann. Führen Sie alle notwendigen Induktionsanfänge sowie Induktionsvoraussetzungen explizit an und *begründen Sie kurz* Ihren Ansatz (insbesondere die Wahl Ihrer Induktionsvariablen). Geben Sie Ihre Lösung als `Induktion.pdf` über EST ab.

Einzelaufgabe 3.2: Längste gemeinsame Teilfolge (LCS) **10 EP**

▶ Seien $X = (x_1, \ldots, x_m)$ und $Y = (y_1, \ldots, y_n)$ zwei Folgen, wobei $x_i, y_j \in Q$ für ein endliches Alphabet Q, dann heißt Y Teilfolge von X, wenn es Indizes $i_1 < i_2 < i_3 < \ldots < i_n$ gibt, mit $x_{i_k} = y_k$ für $k = 1, \ldots, n$.

◇ **Beispiel:** Y = (BCAC) ist Teilfolge von X = (A<u>B</u>A<u>CA</u>B<u>C</u>) – wähle $(i_1, i_2, i_3, i_4) = (2, 4, 5, 7)$

▶ Sind X, Y und Z Folgen über dem Alphabet Q, so heißt Z gemeinsame Teilfolge von X und Y, wenn Z sowohl Teilfolge von X als auch von Y ist.

◇ **Beispiel:** Z = (BCAC) ist gemeinsame Teilfolge von X = (A<u>B</u>A<u>CA</u>B<u>C</u>) und Y = (<u>B</u>A<u>C</u>C<u>A</u>BB<u>C</u>)

▶ Z heißt *längste gemeinsame Teilfolge (LCS)* von X und Y, wenn Z gemeinsame Teilfolge von X und Y ist und es *keine* andere gemeinsame Teilfolge von X und Y gibt, die länger als Z ist.

Erstellen Sie eine Klasse `LaengsteGemeinsameTeilfolge`.

a) Ergänzen Sie die Klasse um eine statische Methode `long[] lgt(long[] x, long[] y)` die *rekursiv* die *längste gemeinsame Teilfolge (LCS)* von `x` und `y` bestimmt.

b) Implementieren Sie nun eine statische Methode `long[] lgt(long[][] z)`, die die *längste gemeinsame Teilfolge (LCS)* aller Felder in `z` ermittelt.

Behandeln Sie `null` als Aktualparameter für `x` oder `y` bzw. `z` wie ein leeres Feld, d.h. das Ergebnis ist dann ebenfalls ein leeres `long`-Feld (*nicht* `null` – werfen Sie *keine* Ausnahme)! Sie dürfen `Arrays.copyOf` aus der Java-API verwenden.
Geben Sie `LaengsteGemeinsameTeilfolge.java` über EST ab.

Gruppenaufgabe 3.3: Rekursionsformen **12 GP**

Legen Sie jeweils fest, welche Rekursionsformen in den folgenden Code-Ausschnitten vorliegen **und begründen** Sie Ihre Antwort. Unterscheiden Sie dabei ggf. explizit zwischen Endrekursion und allgemeiner linearer Rekursion ohne Endrekursion.

a)

```
public int sumDouble(int n) {
    int sum = 0;

    if(n <= 1)
        return 1;

    if(n % 2 == 0)
        sum += n + sumDouble(n - 1);
    else
        sum += 2 * n + sumDouble(n - 2);

    sum += sumDouble(n - 3);

    return sum;
}
```

b)

```
public int sumEven(int n) {
    if(n % 2 == 0)
        return 2 * n + sumOdd(n - 1);
    else
        return sumOdd(n);
}

public int sumOdd(int n) {
    if(n <= 1)
        return 1;
    else
        return n + sumEven(n - 1);
}
```

c)

```
public double sumHalf(int n) {
    if(n <= 1)
        return 1;

    return 0.5 * n + sumHalf(n - 1);
}
```

d)

```
public int sum(int n, int s) {
    if(n <= 1)
        return s + 1;

    return sum(n - 1, s + n);
}
```

e)

```
public int sumRecursive(int n) {
    int sumRecursive = 0;
    int lastSum = n;

    if(n <= 1)
        sumRecursive = 1;
    else {
        for(int i = 0; i <= n; i++) {
            sumRecursive = lastSum + n;
            lastSum = sumRecursive;
        }
    }

    return sumRecursive;
}
```

f)

```
public int sumPart(int n, int m) {
    if(n + m <= 1) return 1;

    return n - sumPart(n / 2,
                  sumPart(m / 2, 0));
}
```

Geben Sie Ihre Lösung als `Rekursion.pdf` über *EST* ab.

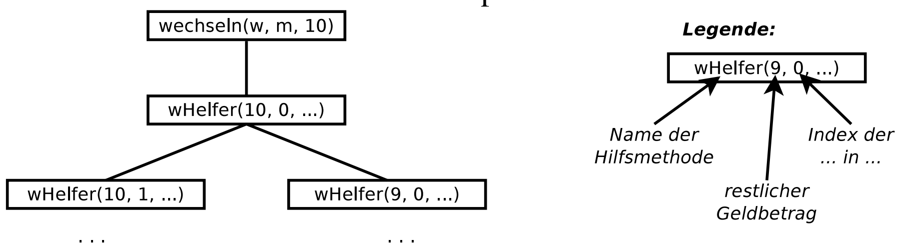
Gruppenaufgabe 3.4: Geld wechseln **15 GP**

Angenommen, eine Währungszone hat k verschiedene Münzbeträge (z.B. gibt es in der Euro-Zone Münzen zu 1^{ct}, 2^{ct}, 5^{ct}, 10^{ct}, 20^{ct}, 50^{ct}, 100^{ct} [= 1€], 200^{ct} [= 2€]). In dieser Aufgabe geht es um die verschiedenen Möglichkeiten, einen beliebigen Geldbetrag (gegeben in der jeweils kleinsten Münzeinheit) in Münzen einer gegebenen Währungszone zu wechseln.
Beispiel: $4^{ct} = 2^{ct} + 2^{ct} = 2^{ct} + 1^{ct} + 1^{ct} = 1^{ct} + 1^{ct} + 1^{ct} + 1^{ct}$

a) Erstellen Sie eine Klasse `GeldWechseln` mit einer öffentlichen statischen Methode namens `void wechseln(Wechsel w, int[] m, int b)`. Das übergebene Feld `m` enthält die möglichen Münzwerte in *beliebiger Reihenfolge* (!) und `b` ist der zu wechselnde Geldbetrag.

Jede gefundene Wechselmöglichkeit `wm` soll als `int`-Feld der Länge `b` (z.B. `{2,2,0,0}` für das obige Beispiel mit $4^{ct} = 2^{ct} + 2^{ct}$ bzw. siehe vorgegebene Testfälle) mittels `Wechsel w` erfasst werden – rufen Sie dafür zum richtigen Zeitpunkt jeweils `w.merke(wm)` auf.
Sie müssen *genau eine* ***rekursive*** Hilfsmethode verwenden, die Sie aus `wechseln` heraus aufrufen (*keine* wechselseitige Rekursion! `wechseln` selbst darf *nicht* rekursiv sein). Überlegen Sie sich das zugrunde liegende Induktionsprinzip zur Konstruktion dieser Methode. Jede Wechselmöglichkeit muss *genau einmal* vorkommen (*keine Duplikate*!).
Achtung: Prüfen Sie die übergebenen Parameter: Wenn sie ungültig sind oder keine Lösung ermöglichen, dann darf die Methode `Wechsel.merke` niemals aufgerufen werden (d.h. es gibt keine Wechselmöglichkeit) – werfen Sie *keine* Ausnahme!

b) Geben Sie für das Beispiel $b = 5^{ct}$ in der Euro-Zone (d.h. `m = {1, 2, 5, 10, 20, 50, 100, 200}`) den Aufrufbaum der Methode `wechseln` bzw. Ihrer Hilfsmethode bis einschließlich `w.merke` an, wobei jeder Knoten auch mit denjenigen Aktualparametern der aufgerufenen Methoden beschriftet sein soll, die sich von Aufruf zu Aufruf ändern und für die Steuerung des Ablaufs (Basisfall vs. Rekursion) relevant sind. Beschreiben Sie kurz in einer Legende, wie der Knoten zu lesen ist – zum Beispiel:



Geben Sie `GeldWechseln.java` bzw. `GeldWechseln.pdf` über EST ab.

Gruppenaufgabe 3.5: Die Türme von Hanoi - Reloaded — 10 GP

Aus den Vorlesungsfolien kennen Sie bereits die „*Türme von Hanoi*“. Die folgende Variante erweitert den Code um Anweisungen zum Zählen der insgesamt erforderlichen Schritte, wobei ein Schritt im Umlegen einer Scheibe von einem Turm auf einen anderen besteht:

```
static int hanoi(int scheiben, char start, char ziel, char hilfe) {
    if (scheiben > 0) {
        int n = 1;
        n += hanoi(scheiben - 1, start, hilfe, ziel);
        System.out.println("Versetze␣Scheibe␣" + scheiben +
                           "␣von␣" + start + "␣nach␣" + ziel);
        n += hanoi(scheiben - 1, hilfe, ziel, start);
        return n;
    } else {
        return 0;
    }
}
```

Beweisen Sie *formal* mittels *vollständiger Induktion*, dass zum Umlegen von k Scheiben (z.B. vom Turm `A` zum Turm `B`) insgesamt $2^k - 1$ Schritte notwendig sind, also dass für $k \geq 0$ gilt:

$$\texttt{hanoi(k, 'A', 'B', 'C')} = 2^k - 1$$

Führen Sie alle notwendigen Induktionsanfänge sowie Induktionsvoraussetzung explizit an und *begründen Sie kurz* Ihren Ansatz. Geben Sie Ihren Beweis als `Hanoi.pdf` über EST ab.

23 EP + 37 GP = 60 Punkte