FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

5. Übung

Abgabe bis 05.06.2017, 10:00 Uhr

Einzelaufgabe 5.1: $\mathcal{O}$-Kalkül – Rechenregeln — 6 EP

Ermitteln Sie für die folgenden Aufwände die **kleinste** passende Aufwandsklasse im $\mathcal{O}$-Kalkül. Hierbei soll n die Eingabegröße sein. Variablen außer n werden als Konstante angesehen.

a) $\mathcal{O}(3 \cdot 2^n + 2 \cdot n^3)$

b) $\mathcal{O}(3n + 6n^2)$

c) $\mathcal{O}(4n^2 \cdot 2 \log n + 3^a)$

d) $\mathcal{O}(n \log^2 n + n^2)$

e) $\mathcal{O}(6\sqrt[a]{n} + 5\sqrt[b]{n})$

f) $\mathcal{O}(\sqrt{n} + \log n^{(\frac{3}{2})})$

g) $\mathcal{O}(n! + 2^n)$

Geben Sie Ihre Lösung als `OKalkuel.pdf` über EST ab.

Einzelaufgabe 5.2: $\mathcal{O}$-Kalkül für Methoden — 12 EP

Geben Sie zu jeder der folgenden Methoden die *kleinste* obere Schranke im $\mathcal{O}$-Kalkül für die Laufzeit so an, dass sich das Ergebnis nicht mehr weiter vereinfachen lässt. Betrachten Sie dabei die Methodenargumente als „Problemgröße“ und nehmen Sie vereinfachend an, dass der Datentyp der verwendeten Variablen unbeschränkt ist sowie grundsätzlich keine Überläufe auftreten können. Geben Sie *jeweils* eine kurze *Begründung* für Ihre Einschätzung an.

a)

```
public int[][] methA(int[][] img) {
    for (int i = 0; i < img.length; i++) {
        for (int j = 0; j < img[i].length; j++) {
            img[i][j] = j;
        }
    }
    return img;
}
```

b)

```
public int methB(int n) {
    int r = 0;
    for (int i = 1; i * 2 * i < 8 * n; i++) {
        r = (int) (r * r / i);
        r++;
    }
    return r;
}
```

c)

```
public int methC(int n) {
    int a = 1;
    int r = 1;
    for (int i = 0; i < n; i++) {
        a = a * 3;
        r = r * a;
    }
    for (int i = 0; i <= a; i++) {
        r--;
    }
    return r;
}
```

d)

```
public int methD(int n) {
    int r = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            for (int k = 0; k <= j; k++) {
                r += i * j * k;
                return r;
            }
        }
    }
    return r;
}
```

e)

```
public int methE(int n) {
    int r = 0;
    for (int i = 1; i < n; i++) {
        r += n;
        n = n / 3;
    }
    return r;
}
```

f)

```
public static int methF(int n) {
    int r = 1;
    for (int i = 0; i < n; i++) {
        r += methF(n - 1);
    }
    return r;
}
```

Geben Sie Ihre Lösung als `Laufzeitanalyse.pdf` über EST ab.

Einzelaufgabe 5.3: Stapel — 9 EP

Gegeben ist folgende Java-Klasse:

```
1  class Zahl {
2      private int value;
3
4      public Zahl(int v) {
5          value = v;
6      }
7
8      public Zahl add(Zahl i) {
9          value += i.value;
10         return this;
11     }
12
13     public Zahl acc(Zahl i) {
14         return new Zahl(value + i.value);
15     }
16     public Zahl meth(Zahl i, Zahl j) {
17         int k = 23;
18         Zahl t = new Zahl(k);
19         i = i.acc(t);
20         j = j.add(i);
21         return j;
22     }
23
24     public void test() {
25         int k = 42;
26         Zahl i = new Zahl(k);
27         Zahl j = new Zahl(5);
28         i = meth(i, j);
29     }
30 }
```

Untersuchen Sie für die Ausführung der Methode `test` die Programmstapel- und die Speicherbelegung. Die Belegung vor dem Aufruf von `meth` in Zeile 28 ist in der Abbildung 1 gegeben. Im Stapel sind die Variablennamen angegeben und je nach Variablentyp der Wert oder die Speicheradresse (mit # gekennzeichnet) des Objekts. Im Speicher sind für alle Felder der gespeicherten Objekte die Werte angegeben (Referenzen sind in diesem Beispiel nicht möglich).
Überlegen Sie, wie sich durch den Aufruf von `meth` die Belegung von Stapel und Speicher ändert. Zeichnen Sie dazu jeweils den Zustand nach dem Ausführen der Zeilen 20 und 28 im Format, das im Beispiel vorgegeben ist. Nehmen Sie an, dass der Garbage-Collector zu keinem Zeitpunkt aktiv ist. Der tatsächliche Speicherverbrauch soll ignoriert werden; jeder Typ verbraucht eine Speicherstelle. Die Speicherstellen sollen aufsteigend vergeben werden. Geben Sie `Stapel.pdf` im EST ab.

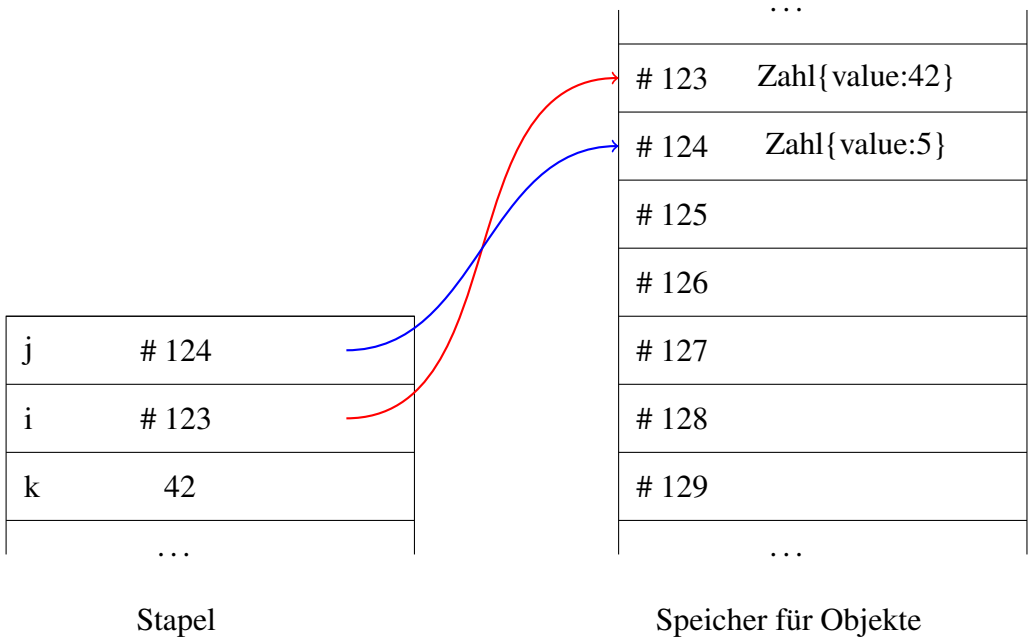

Abbildung 1: Belegung vor dem Aufruf der Methode `meth`.

Gruppenaufgabe 5.4: Die Matrix – jetzt auch objekt-orientiert — 33 GP

In dieser Aufgabe sollen Sie eine Klasse `Matrix` für ganzzahlige Matrizen inklusive zugehöriger Methoden für typische Matrix-Operationen nach folgenden Vorgaben implementieren. *Beachten* Sie unbedingt, dass in dieser Implementierung alle *Zeilen und Spalten Java-typisch* 0*-indiziert* sind.

`Matrix(int m):` Dieser Konstruktor erzeugt eine $(|m| \times |m|)$-Matrix, deren Zellen mit 0 gefüllt sind. Der Konstruktor muss das Argument prüfen und ggf. korrigieren: Ist m negativ, dann wird stattdessen der positive Betrag $|m|$ als Dimension verwendet.

`Matrix(int m, int n):` Dieser Konstruktor verhält sich analog zu `Matrix(int m)`, erzeugt aber eine $(|m| \times |n|)$-Matrix mit $|m|$ Zeilen und $|n|$ Spalten.

`Matrix(Matrix andere):` Dieser Konstruktor erzeugt eine *Kopie* der übergebenen Matrix `andere`. Auch hier sind ggf. Parameterprüfungen notwendig: Ist `andere` `null`, dann soll stattdessen eine leere (0×0)-Matrix konstruiert werden.

`int anzahlZeilen()` bzw. `anzahlSpalten()`: Diese beiden Methoden geben ihrem Namen entsprechend die Anzahl m der Zeilen bzw. n der Spalten der aktuellen Matrix $\mathcal{A}$ zurück. Die leere (0×0)-Matrix habe dabei 0 Spalten.

`void setzeWert(int i, int j, long w)`: Setzt den Wert a_{ij} der aktuellen Matrix $\mathcal{A}$ auf w, sofern die Indizes i und j gültig sind – andernfalls bleibt $\mathcal{A}$ unverändert.

`long holeWert(int i, int j)`: Gibt den Wert a_{ij} der aktuellen Matrix $\mathcal{A}$ zurück, sofern die Indizes i und j gültig sind – andernfalls gibt die Methode 0 zurück.

`Matrix addiere(Matrix a, Matrix b)`: Diese Klassenmethode führt die Matrizenaddition $c := a + b$ aus und gibt c zurück, sofern a und b von `null` verschieden sind und passende Dimensionen haben – andernfalls gibt sie ebenfalls `null` zurück. Die Eingaben a und b dürfen dabei *nicht* verändert werden. Sie müssen die Einhaltung des Wertebereichs der einzelnen Zellen hier nicht überprüfen.

`Matrix multipliziere(Matrix a, Matrix b)`: Verhält sich im Wesentlichen analog zu `addiere`, führt jedoch stattdessen die Matrizenmultiplikation $c := a \cdot b$ aus.

`void multipliziere(long l)`: Führt eine Skalarmultiplikation der aktuellen Matrix $\mathcal{A}$ mit dem Skalar λ „in-place“ durch, d.h. der Zustand des Objekts *muss* sich diesmal verändern.

`Matrix transponiere()`: Berechnet die Transponierte Matrix $\mathcal{A}^T$ der aktuellen Matrix $\mathcal{A}$ und gibt $\mathcal{A}^T$ zurück. Die Original-Matrix $\mathcal{A}$ bleibt dabei unverändert erhalten!

`Matrix unterMatrix(int i, int j)`: Ermittelt die Untermatrix $\mathcal{A}_{ij}$ der aktuellen Matrix $\mathcal{A}$. Sie entsteht aus $\mathcal{A}$ durch Weglassen der Zeile i und der Spalte j, sofern diese jeweils existieren (getrennte/individuelle Prüfung!). Die Original-Matrix $\mathcal{A}$ bleibt dabei unverändert erhalten!

`long determinante()`: Berechnet die Determinante der aktuellen Matrix $\mathcal{A}$, sofern diese quadratisch $(m \times m)$ ist, andernfalls gibt sie 0 zurück. Beachten Sie auch die „Basisfälle“ einer leeren (0×0) oder einwertigen (1×1) Matrix.

Achtung - Wichtig: Sie dürfen hier **keinerlei** Methoden oder Klassen aus der Java-API verwenden! Geben Sie Ihre Implementierung in der Datei `Matrix.java` über EST ab.

27 EP + 33 GP = 60 Punkte