HSLU Hochschule Luzern

Algorithmen & Datenstrukturen (AD)

Übung: Einführung E1

Themen: Einführung: Algorithmen, Datenstrukturen, Komplexität, Rekursion.
Zeitbedarf: ca. 5 Stunden

Ingmar Baetge, Hansjörg Diethelm, Version 2.6 (HS 2025)

1 Zum Einstieg (ca. 30')

1.1 Lernziel

- zum Motivieren und Aufwärmen

1.2 Grundlagen

Diese Aufgabe basiert auf dem Input E11 und beinhaltet keine Programmierung.

1.3 Aufgaben

a) Optional: Gehen Sie auf die Webseite https://de.wikipedia.org/wiki/Liste_von_Algorithmen und verschaffen Sie sich in 15' einen kleinen Einblick in die vielfältige Welt der Algorithmen.

b) Wir haben drei unterschiedliche Implementationen des euklidischen Algorithmus angeschaut: **`ggtIterativ1(…)`**, **`ggtIterativ2(…)`** sowie **`ggtRekursiv(…)`**. Durchlaufen Sie die drei Algorithmen für die Berechnung des ggT von (16, 68) "von Hand auf Papier". Versuchen Sie's mit einer gut nachvollziehbaren Darstellung. Natürlich sollte überall dasselbe Ergebnis resultieren. Wie viele Schleifendurchläufe ergeben sich für die beiden iterativen Varianten? Wie viele Methodenaufrufe erfolgen im rekursiven Fall? Was sagen Sie betreffend Speicherbedarf zu den drei Implementationen?

2 Laufzeit-Betrachtungen (ca. 45')

2.1 Lernziele

- Das Laufzeitverhalten via Anzahl Methodenaufrufe bzw. via Laufzeitmessungen bestimmen/verifizieren können.
- Erkennen, dass in diesem Beispiel der Zeitaufwand für die Schleifensteuerungen tatsächlich vernachlässigt werden kann.

2.2 Grundlagen

Diese Aufgabe basiert auf dem Input E11.

2.3 Aufgaben

a) Implementieren Sie das Beispiel **`task(…)`** aus dem Input (Folie 26). Die Methoden **`task1(…)`** bis **`task3(…)`** können Sie vorerst mit leeren Bodies programmieren. Modifizieren Sie dann den Code so, dass am Ende die Gesamtanzahl der Methodenaufrufe von **`task1(…)`** bis **`task3(…)`** auf der Konsole ausgegeben wird (z.B. mit SLF4J oder Log4J).

   Machen Sie einige Experimente, indem Sie **`task(…)`** mit unterschiedlichen n aufrufen und die resultierende Anzahl Methodenaufrufe in einer Tabelle festhalten. Letztere sind ein Mass für die Laufzeit.

b) Für genügend grosse n sollte sich bei einer Verdoppelung eine Vervierfachung der Aufrufe ergeben. Stellen Sie dies auch tatsächlich fest?

c) Sie können das Beispiel noch realistischer umsetzen, indem Sie die Methoden **`taskX(…)`** für eine bestimmte Zeit "schlafen legen", z.B. alle für 5ms. Die Methoden **`Thread.sleep(…)`** und **`System.currentTimeMillis(…)`** sind in diesem Zusammenhang hilfreich. Entscheidend für das resultierende Laufzeitverhalten der Methode **`task(…)`** ist, dass die Methoden **`taskX(…)`** je O(1) besitzen. Im Prinzip müssen also nicht alle genau dieselbe Lauf- bzw. Sleep-Zeit aufweisen. Wichtig ist einzig, dass sie konstant sind.

   Sie sollten dann die Feststellung machen, dass sich für genügend grosse n die Laufzeit quadratisch verhält. Halten Sie dazu die gemessenen Laufzeiten ebenfalls in einer Tabelle fest.

3 Ordnung (ca. 45')

3.1 Lernziele

- Ein Gefühl für die unterschiedlichen Wachstumsfunktionen bekommen.
- Die Ordnung für einfache Funktionen bestimmen können.
- Aufgrund der Ordnung grobe Zeitabschätzungen machen können.
- Die Ordnung aus einfachen Programmstrukturen ableiten können.

3.2 Grundlagen

Diese Aufgabe basiert auf dem Input E11 und beinhaltet keine Programmierung.

3.3 Aufgaben

a) Vervollständigen bzw. berechnen Sie eine Tabelle der folgenden Art, um einen Eindruck der verschiedenen Wachstumsfunktionen zu bekommen:

log n	ld n	**n**	n·log n	n^2	n^3	2^n	3^n	n!
		1						
		2						
		5						
		10						
		20						
		50						
		100						

b) Bestimmen Sie von folgenden Funktionen deren Ordnung:

(1) $f(n) = n^3 + 0.1 \cdot 2^n$

(2) $f(n) = 5326 + \ln(n)$

(3) $f(n) = 2 + 37 \cdot n^3 + 0.01 \cdot n^4$

(4) $f(n) = 1000 \cdot n + n^3$

(5) $f(n) = n^7 + n!$

(6) $f(n) = \ln(n) + 1000 \cdot n$

c) Ordnen Sie die resultierenden Ordnungen von b) gemäss ihrer Wachstumsrate, d.h. zuerst die kleinste, dann aufsteigend die grösseren.

d) Die Rechenzeit t betrage jeweils 0.1s, und zwar bei drei verschiedenen Algorithmen im Falle von n = 10'000. Für dieses n verhalten sich die drei Algorithmen gemäss ihrer Ordnung. Geben Sie an, was für Rechenzeiten für n = 100'000 etwa zu erwarten sind?

(1) $O(1)$

(2) $O(n^2)$

(3) $O(n \cdot \log n)$

e) Zeigen Sie auf, weshalb bei $O(\log_B n)$ der Wert der Basis keine Rolle spielt.
Tipp: Wie lässt sich schon wieder der Logarithmuswert für eine beliebige Basis B ausrechnen? Schauen Sie sich gegebenenfalls das allerletzte Slide im Input E11 an.

f) Bestimmen Sie die Ordnung bezüglich Laufzeit von nachfolgenden Methoden. Hinweise:

- Die Methoden werden jeweils mit $n \geq 0$ aufgerufen.
- Die Hilfsmethoden `doA()` und `doB()` besitzen je die Ordnung O(1).
- Die Hilfsmethode `doC(n)` besitzt die Ordnung O(n).
- Die Hilfsmethode `doD(n)` besitzt die Ordnung $O(n^2)$.

(1)

```
public static void exercise1(final int n) {
    doA();
    doB();
    doC(n);
}
```

(2)

```
public static void exercise2(final int n) {
    for(int i=0; i < n; i++) {
        doA();
    }
    for(int j=0; j < (2 * n); j++) {
        doC(5);
    }
}
```

(3)

```
public static void exercise3(final int n) {
    for(int i=0; i < n; i++) {
        doA();
        for(int j=0; j < n; j++) {
            doC(n);
        }
    }
}
```

(4)

```
public static void exercise4(final int n) {
    for (int i = 0; i < n; i++) {
        doD(n);
        for (int j = 0; j < (n + 5); j++) {
            doA();
        }
    }
}
```

4 Fibonacci-Zahlen (ca. 60')

4.1 Lernziele

- Eine einfache rekursive Methode implementieren können.
- Rekursionsbasis und Rekursionsvorschrift identifizieren können.
- Erkennen, dass man eine rekursive Methode allenfalls einfach optimieren kann, oder die Rekursion sogar einfach durch eine Iteration ersetzen kann.
- Erkennen, dass man häufig die Laufzeit zu Lasten von mehr Speicher verkürzen kann.

4.2 Grundlagen

Diese Aufgabe basiert auf dem Input E12.

4.3 Aufgaben

a) Implementieren Sie "straight forward" gemäss Input eine Methode **`fiboRec1(…)`**. Testen Sie die Methode mit Hilfe von JUnit, indem Sie mindestens 3 sinnvolle Testfälle programmieren. Es gelte $n \geq 0$.

b) Wo in Ihrem Code machen Sie die Rekursionsbasis und die Rekursionsvorschrift aus? Kommentieren Sie den Code entsprechend. Die Bedingungen gehören dazu.

c) Bekanntlich ist die Implementation gemäss a) suboptimal, weil insbesondere wiederholt dieselben Zwischenresultate berechnet werden! Bezüglich Laufzeit wäre optimaler, Zwischenresultate zu speichern, um darauf ohne erneutes Berechnen wieder zugreifen zu können. Programmieren Sie eine entsprechende Methode **`fiboRec2(…)`**. Ein Lösungsansatz ist, ein Array als Klassenvariable einzuführen. Probieren Sie's. Die modifizierte Methode bedingt nur wenige Anpassungen. Die bringen's aber!

d) Die Fibonacci-Zahlen rekursiv zu berechnen ist zwar naheliegend und elegant, andererseits geht es aber auch iterativ relativ einfach. Programmieren Sie eine entsprechende Methode **`fiboIter(…)`**.

e) Optional: Vergleichen Sie die Laufzeiten der drei Methoden, indem Sie die gemessenen Laufzeiten in einer Tabelle festhalten.

4.4 Anmerkung

Da mag man staunen! Formel von Moivre-Binet:

$$f_n = \frac{1}{\sqrt{5}} \cdot \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

So bekommt man z.B. für $f_{14} = 377$

Entsprechend kann man f_n für grosse n näherungsweise wie folgt berechnen:

$$f_n \approx \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2} \right)^n$$

So bekommt man z.B. für $f_{14} = 377.00053$

5 Ackermann-Funktion (ca. 45')

5.1 Lernziele

- Ein Aha-Erlebnis haben und Respekt bekommen. Sieht so harmlos aus, aber …
- Genau verstehen und nachvollziehen können, wie rekursive Methoden abgearbeitet werden.

5.2 Grundlagen

Diese Aufgabe basiert auf dem Input E12.

Die Ackermann-Funktion **`ack(…)`** ist wie folgt rekursiv definiert:

(i) ack(0, m) = m + 1

(ii) ack(n, 0) = ack(n-1, 1) für n > 0

(iii) ack(n, m) = ack(n-1, ack(n, m-1)) für n, m > 0

Diese Funktion kann man als "Atombombe" in der Informatik bezeichnen! Bereits für kleine Parameter-Werte von n und m resultieren astronomische Funktionswerte, obschon nur "+1" gerechnet wird. Damit verbunden erfolgen astronomisch viele Methoden-Aufrufe, die jeden Rechner bzw. Call Stack zum Crash führen. Es ist also Vorsicht am Platz. Die Ackermann-Funktion ist primär "Gegenstand" der theoretischen Informatik.

5.3 Aufgaben

a) Die Ackermann-Funktion ist ein Beispiel für ein Problem, das nicht primitiv rekursiv ist. Woran erkennt man dies? Die Antwort finden Sie im Input bei den Rekursions-Typen.

b) Berechnen Sie von Hand auf Papier für folgende Parameter-Kombinationen von n und m die Funktionswerte der Ackermann-Funktion. Versuchen Sie dabei, die Rekursionsaufrufe gut nachvollziehbar aufzuzeichnen.

n:	0	1	2
m:	4	2	2

Aufgepasst(**!!**): ack(4, 2) ist bereits $2^{65536} - 3$.

c) Wie viele Methodenaufrufe erfolgen insgesamt beim Berechnen von ack(2, 2)?

d) Optional: Wie hoch wird der Call Stack bei der Berechnung von ack(2, 2) im Maximum, d.h. wie viele Frames liegen in diesem Moment auf dem Stack?

Beachte: Auf Ebene JVM werden die Parameter einer Methode vorweg berechnet bzw. ausgewertet!

Java Source Code	ack(1, ack(2, 1))
→ JVM	int **parameter2** = ack(2, 1); ack(1, **parameter2**);

e) Optional: Programmieren Sie eine entsprechende Methode **`ack(…)`**. Dafür benötigen Sie keine fünf Minuten. Es empfiehlt sich, mindestens mit dem Datentyp **`long`** zu arbeiten. Aber aufgepasst, rufen Sie die Methode nur für ganz, ganz kleine Parameter-Kombinationen auf (siehe oben)! Googeln Sie mal, wie sich die Grösse des Call Stacks bei Java konfigurieren lässt.

6 Färben in einem pixel-basierten Zeichnungsprogramm (ca. 30')

6.1 Lernziele

- Genau verstehen und nachvollziehen können, wie rekursive Methoden abgearbeitet werden.
- Ein schönes Beispiel aus der Computergrafik kennenlernen.

6.2 Grundlagen

Diese Aufgabe basiert auf dem Input E12 und beinhaltet keine Programmierung.

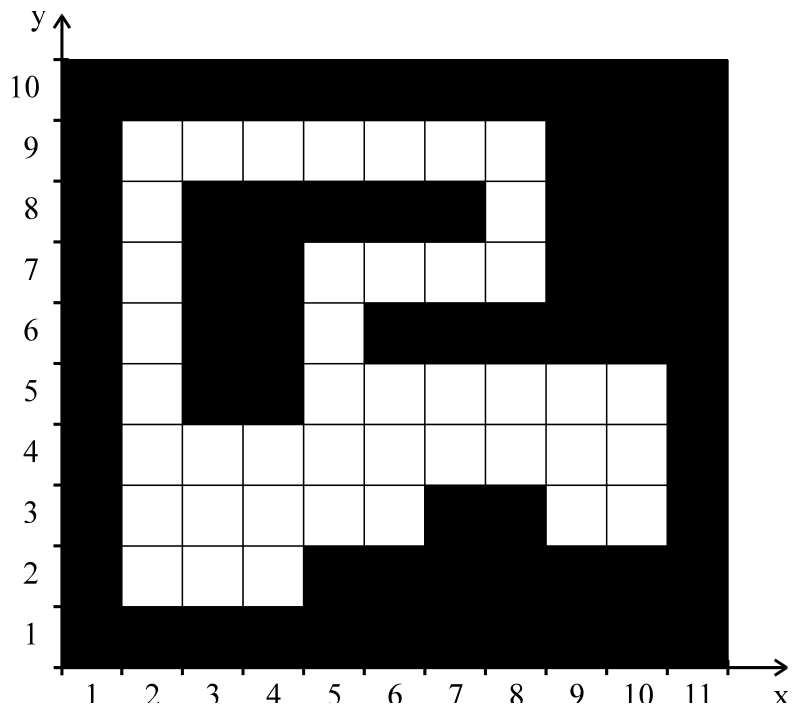
6.3 Aufgaben

Die Methode **`colorArea(…)`** füllt bzw. färbt eine Fläche pixelweise mit der Farbe **`fillColor`**. Die Ausdehnung der zu färbenden Fläche ergibt sich durch die andere Umgebungsfarbe **`outsideColor`**. Aufgerufen wird die Methode mit den Koordinaten (**`x, y`**) eines zu färbenden Pixels. Man leert also sozusagen an der Stelle (**`x, y`**) den "Farbkübel" aus.

```
public void colorArea(final int x,
                      final int y,
                      final Color fillColor,
                      final Color outsideColor) {
    Color actualColor;
    actualColor = getActualColor(x, y);
    if ((actualColor != outsideColor) && (actualColor != fillColor)) {
        colorPixel(x, y, fillColor);
        colorArea(x + 1, y, fillColor, outsideColor);
        colorArea(x, y + 1, fillColor, outsideColor);
        colorArea(x - 1, y, fillColor, outsideColor);
        colorArea(x, y - 1, fillColor, outsideColor);
    }
}
```

a) Nummerieren Sie in folgendem Bild die Pixel gemäss der Reihenfolge, wie sie von "weiss" auf "grau" gefärbt werden. Die zu färbende Fläche wird durch die "schwarze" Umgebung definiert. Die Methode wird wie folgt aufgerufen:

```
colorArea(4, 3, Color.gray, Color.black);
```



b) Falls mit `colorPixel(…)` ein Pixel gefärbt wird (vgl. Source-Code), zieht dies unumgänglich vier rekursive Methodenaufrufe von `colorArea(…)` nach sich. Da geht also auch "die Post ab"! Versuchen Sie zumindest ansatzweise aufzuzeichnen, wie der damit verbundene Rekursionsbaum aussieht. Damit's etwas einfacher wird mit Aufschreiben, starten Sie vorteilhaft mit der Kurzschreibweise **`cA(4,3)`** für obigen Methodenaufruf.

c) Wie oft werden ihm obigen Beispiel die Methode `colorPixel(…)` und die Methode `colorArea(…)` je insgesamt aufgerufen?

7 Optional: Algorithmen für symmetrische Zahlen (ca. 45')

7.1 Lernziele

- Selbst Algorithmen entwickeln.
- Einfache Problemstellung vs. unterschiedliche Lösungsverfahren (Lösungs-Prinzipien) entdecken.
- Unterschiedliche Lösungsverfahren vs. adäquate Datentypen/Datenstrukturen anwenden.

7.2 Grundlagen

Eine Zahl ist symmetrisch, wenn sie von links nach rechts gelesen gleich ist, wie von rechts nach links gelesen. Man spricht auch von Zahlenpalindrom. Z.B. sind 242, 7 und 1234321 symmetrische Zahlen; 1234 ist keine. Im Falle von 4-stelligen Zahlen sind z.B. 0110 und 5555 symmetrisch; nicht aber 2420 oder 0242!

7.3 Aufgaben

a) Geben Sie ein paar 3-stellige symmetrische Zahlen an.

b) Wie viele verschiedene 1-stellige symmetrische Zahlen gibt es?
Wie viele verschiedene 2-stellige symmetrische Zahlen gibt es?
Wie viele verschiedene 3-stellige symmetrische Zahlen gibt es?
Wie viele verschiedene 4-stellige symmetrische Zahlen gibt es?

c) Wie viele symmetrische Zahlen gibt es im Falle von n Stellen?

d) Entwickeln Sie einen Algorithmus, welcher für eine vorgegebene n-stellige Zahl entscheidet, ob diese symmetrisch ist (ja/nein). Z.B. n = 7 und 8227228 → ja. Beschreiben Sie Ihren Algorithmus möglichst präzise in Prosa oder mit Pseudo-Code.

e) Implementieren und testen Sie Ihren Algorithmus von d).

f) Entwickeln Sie einen Algorithmus, welcher für eine vorgegebene n-stellige Zahl die nächstgrössere symmetrische Zahl bestimmt. Z.B. n = 3 und 193 → 202. Beschreiben Sie Ihren Algorithmus möglichst präzise in Prosa oder mit Pseudo-Code.

g) Implementieren und testen Sie Ihren Algorithmus von f).

h) Reflektieren Sie Ihre verwendeten Datentypen/Datenstrukturen in e) und g).