AD FS 2020

Victor Fernández

Januar 2020

Inhaltsverzeichnis

Teil I
SW 01 - Einführung Algorithmen, Datenstrukturen & Komplexität

1 Lernziele

Sie ...
- können beschreiben, was ein Algorithmus ist
- können erläutern, was gleichwertige Algorithmen sind
- können erläutern, weshalb Algorithmen und Datenstrukturen eng zusammenhängen
- können beschreiben, was Komplexität bei einem Algorithmus meint
- können für einfache Funktionen deren Ordnung bestimmen
- können für einfache Code-Fragmente deren Zeitkomplexität bestimmen
- kennen die wichtigsten Ordnungsfunktionen im Vergleich
- kennen wichtige Aspekte bei der Interpretation einer Ordnung
- wissen, welche Ordnungen praktisch versagen!

2 Algorithmen

2.1 Definition Algorithmus

- Ein Algorithmus ist ein **präzise festgelegtes Verfahren zur Lösung eines Problems**; genauer gesagt, zur Lösung einer Problem**klasse** (beinhaltend gleichartige Probleme, häufig unendlich viele).
- Algorithmus=Lösungsverfahren (Rezept, Anleitung)
- Probleme bzw. Problemklassen, die mit Algorithmen gelöst werden können, heissen ⇒ **berechenbar**

2.2 Beispiele

- Berechnung des **ggT** für zwei natürliche Zahlen (Euklidischer Algorithmus)
- Zeichnen der **Verbindungslinie**, welche zwei Punkte verbindet (Bresenham Algorithmus)
- **Sortierung** von zufällig vorliegenden ganzen Zahlen (Mergesort Algorithmus)
- Finden des **kürzesten Weges** zwischen zwei Knoten in einem zusammenhängenden Graphen (Algorithmus von Dijkstra)
- Entscheiden, ob es sich bei einer vorliegenden natürlichen Zahl um eine **Primzahl** handelt (Algorithmus „Sieb von Aktin“)
- Berechnung des **Integrals** bei vorliegenden Funktionswerten in einem bestimmten Bereich (Runge-Kutta Algorithmus)
- Finden einer **Lösung** in einem vorgegebenen Lösungsraum (Backtracking Algorithmus)

2.3 Eigenschaften eines Algorithmus

- schrittweises Verfahren
- ausführbare Schritte
- eindeutiger nächster Schritt (**determiniert**)
- endet nach endlich vielen Schritten (**terminiert**)

2.4 Algorithmen vs. Informatik

- Der Computer:
  - arbeitet schrittweise
  - Anweisung für Anweisung (jede Anweisung korrespondiert mit einem ausführbaren Befehl)
  - arbeitet präzise und schnell
- Algorithmen sind zentrales Thema in der Informatik und in der Mathematik
  - **Algorithmentheorie**: Guter Lösungsalgorithmus für bestimmte Problemstellung?
  - **Komplexitätstheorie**: Ressourcenverbrauch von Rechenzeit und Speicherbedarf?
  - **Berechenbarkeitstheorie**: Was ist mit einer Maschine grundsätzlich lösbar und was nicht?

2.5 Algorithmen und Datenstrukturen

- **Algorithmen operieren auf Datenstrukturen** und **Datenstrukturen bedingen spezifische Algorithmen**. Beides ist eng miteinander verbunden.
- Bei vielen Algorithmen hängt der **Ressourcenbedarf**, d.h. die benötigte Laufzeit und der Speicherbedarf, bon der Verwendung geeigneter Datenstrukturen ab.

2.6 Beispiel Euklidischer Algorithmus

2.6.1 Manuelle Ausführung

ggT von 8 und 14:

Schritt	A	B	A-B
1	14	8	6
2	8	6	2
3	6	2	4
4	4	2	2
5	2	2	0=ggT ↑ 2

2.6.2 Iterative Implementation

Beispielcode (1):

```
public static int ggtIterativ(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

2.6.3 Iterative Implementation

Beispielcode (2):

```
public static int ggtIterativ(int a, int b) {
    while ((a != 0) && (b != 0)) {
        if (a > b) {
            a = a % b;
        } else {
            b = b % a;
        }
    }
    return (a + b); // a oder b ist 0
}
```

2.6.4 Rekursive Implementation

```
public static int ggtRekursiv(final int a, final int b) {
    if (a > b) {
        return ggtRekursiv(a - b, b);
    } else {
        if (a < b) {
            return ggtRekursiv(a, b -a);
        } else {
            return a;
        }
    }
}
```

3 Datenstrukturen

3.1 Definition Datenstruktur

Eine Datenstruktur ist ein **Konzept zur Speicherung und Organisation von Daten**. Es handelt sich um eine **Struktur**, weil die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung möglichst effizient zu ermöglichen.
Datenstrukturen sind daher insbesondere auch durch die **Operationen** charakterisiert, welche Zugriff und Verwaltung realisieren.

3.2 Beispiele

- **Array**: direkter Zugriff (+), fixe Grösse (-)
- **Liste**: flexible Grösse (+), sequentieller Zugriff (-)

4 Komplexität

4.1 Definition Komplexität

- Komplexität (auch Aufwand oder Kosten) eines Algorithmus
  - **Ressourcenbedarf = f (Eingabedaten)**
    D.h. „Wie hängt der Ressourcenbedarf von den Eingabedaten ab?“
- Ressourcenbedarf
  - Rechenzeit: **Zeitkomplexität**
  - Speicherbedarf: **Speicherkomplexität**
- Eingabedaten
  - Grösse der Daten**menge** (z.B. 10 vs. 1’000’000’000 zu sortierende Elemente)
  - Grösse eines Daten**wertes** (z.B. 10! vs. 1’000’000’000!)

Was interessiert uns?

- **Wie wächst der Ressourcenbedarf**, wenn eine grössere Datenmenge bzw. grössere Datenwerte zu verarbeiten sind? Z.B.
  - Verdoppelt oder vervierfacht sich der Ressourcenbedarf für das Sortieren der doppelten Datenmenge?
  - Bleibt der Ressourcenbedarf gleich, wenn wir den ggT von zwei sehr grossen Zahlenwerten berechnen wollen?
- Es interessiert an dieser Stelle **NICHT** der exakte/absolute Ressourcenbedarf! Z.B.
  - Die ggT-Berechnung von 1’000’000’489 und 9’123’000’124 auf dem Computer XY mit der Konfiguration Z dauert 420ms.
  - Entsprechende Rechenzeiten sind für jeden Computer anders. Möchte man die Rechenzeit reduzieren, so lässt sich jederzeit ein schnellerer Computer kaufen!

4.1.1 Zeitkomplexität Implementation

- Annahmen:
  - Die Methoden `task1()`, `task2()` und `task3()` besitzen in etwa dieselben Rechenzeiten
  - Die Schleifensteuerungen beanspruchen im Vergleich vernachlässigbare kleine Ausführungszeiten

```
1 public static void task(final int n) {
2     task1(); task1(); task1(); task1();      // T ~ 4
3     for (int i = 0; i < n; i++) {            // äussere Schleife: n-mal
4         task2(); task2(); task2();           // T ~ n · 3
5         for (int j = 0; j < n; j++) {        // innere Schleife: n-mal
6             task3(); task3();                // T ~ n · n · 2
7         }
8     }
9 }
```

→ Rechenzeit T von `task(n)`: $T = f(n) \sim 4 + 3 \cdot n + 2n^2$

4.1.2 Zeitkomplexität für grosse n

TODO Tabelle Zeitkomplexität
- Für grosse n dominiert der Anteil von n^2

- Für grosse n verlaufen die Funktionen parallel, d.h. unterscheiden sich nur durch einen konstanten Faktor (vgl. logarith. Massstäbe!)
- Wir sagen:
  - f(n) ist von der **Ordnung** $O(n^2)$ bzw. die
  - Rechenzeit von `task(n)` verhält sich gemäss **Ordnung** $O(n^2)$

TODO Big-O & Ordnungsfunktionen

Teil II
Rekursion

5 Lernziele

Sie ...
- können beschreiben, was Algorithmen und Datenstrukturen mit Selbstähnlichkeit und Selbstbezug zu tun haben
- können bei einer rekursiven Methode Rekursionsbasis und Rekursionsvorschrift identifizieren
- können gut nachvollziehbar aufzeichnen, wie eine rekursive Methode abgearbeitet wird
- können beschreiben, wozu „Heap“ und „Call Stack“ dienen
- können die Eigenheiten der Rekursion (vs. Iteration) beschreiben
- können einfache rekursive Methoden implementieren

6 Iteration vs. Rekursion

6.1 Iterative Fakultätsberechnung

6.2 Rekursive Fakultätsberechnung

7 Call Stack

- Für die Ausführung eines Programmes verwendet die Java Virtual Machine (JVM) zwei wichtige Speicher: **Heap** und **Call Stack**
- **Heap**: In diesem Speicherbereich werden die Objekte gespeichert, d.h. deren Instanzvariablen bzw. Zustände. Nicht mehr referenzierbare Objekte werden durch den Garbage Collector (GC) automatisch gelöscht.
- **Call Stack**: Letztendlich wid bei der Ausführung eines Java-Programmes eine Kette von Methoden aufgerufen, bzw. abgearbeitet. Ursprung ist die `main()`-Methode. Jeder Methodenaufruf bedingt gewissen Speicher, insbesondere für die aktuellen Parameter und lokalen Variablen. Dazu dient der Call Stack. Ein neuer Methodenaufruf bewirkt, dass der Call Stack wächst, bzw. darauf ein zusätzlicher **Stack Frame** angelegt wird.

TODO Bild Call Stack 7

7.1 Mächtigkeit der Rekursion

- Rekursion und Iteration sind praktisch **gleich mächtig**
- D.h. die Menge der berechenbaren Problemstellungen bei Verwendung der Rekursion und Verwendung der Iteration ist gleich
- D.h. eine rekursive Implementation lässt sich grundsätzlich immer in eine gleichwertige iterative Implementation umprogrammieren und umgekehrt

Hinweis: Dies gilt exakt nur für sogenannte primitiv-rekursive Probleme, d.h. bei linearer und nicht geschachtelter Rekursion bzw. bei reinen Zählschleifen.