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 $\Rightarrow$ **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):

```java
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):

```java
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

```java
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

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

→ 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.

Teil III
SW 02 - Datenstrukturen

8 Lernziele

- Sie kennen Eigenschaften von Datenstrukturen
- Sie können die Komplexität von Operationen auf unterschiedlichen Datenstrukturen beurteilen
- Sie kennen den Aufbau, die Eigenschaften und die Funktionsweise ausgewählter Datenstrukturen
- Sie können Datenstrukturen exemplarisch selbst implementieren
- Sie können abhängig von Anforderungen die geeigneten Implementationen von Datenstrukturen auswählen

8.1 Eigenschaften von Datenstrukturen

- In welcher Reihenfolge oder Sortierung werden die Elemente Abgelegt
- Welche Operationen werden zur Verfügung gestellt
- Ist die Datenstruktur dynamisch oder statisch (Grösse)
- Bestehen zwischen den Elementen explizite oder implizite Beziehungen
- Besteht direkter oder nur indirekter/sequenzieller Zugriff auf die einzelnen Datenelemente
- Wie gross ist der Aufwand für die einzelnen Operationen, speziell in Abhängigkeit zur Datenmenge

8.2 Reihenfolge uns Sortierung

- Datenstrukturen als reine Sammlung: Die einzelnen Datenelemente sind darin ungeordnet abgelegt und die Reihenfolge ist nicht deterministisch
  - Analogie: Steinhaufen
- Datenstrukturen welche die Datenelemente in einer bestimmten Reihenfolge (z.B. in der Folge des Einfügens) enthalten und diese implizit beibehalten
  - Analogie: Stapel von Büchern
- Datenstrukturen welche die Elemente (typisch beim Einfügen) implizit sortieren / ordnen
  - Analogie: Vollautomatisches Hochregal
- Auch abhängig von der Implementation bzw. Nutzung

8.3 Operation auf Datenstrukturen

- Es gibt einige elementare Methoden die auf Datenstrukturen angewendet werden können
  - Einfügen von Elementen
  - Suchen von Elementen
  - Entfernen von Elementen
  - Ersetzen von Elementen

  in Datenstrukturen
- Operation in Abhängigkeit einer (optionalen) Reihenfolge oder Sortierung (natürlich oder speziell)
  - Nachfolger: Nachfolgendes Datenelement
  - Vorgänger: Vorangehendes Datenelement
  - Sortierung: Sortieren der Datenelemente nach Attributwerten
  - Maxima und Minima: kleinstes und grösstes Datenelement

8.4 Statische vs. dynamische Datenstruktur

- Eine **statische** Datenstruktur hat nach ihrer Initialisierung eien feste, unveränderliche Grösse. Sie kann somit nur eine beschränkte Anzahl Elemente aufnehmen.
  - Analogie: Getränkeflasche
    - Grösse der Flasche ist gegeben, ebenso maximaler Inhalt
    - Die Flasche selber nimmt immer denselben Platz ein
- Eine **dynamische** Datenstruktur hingegen kann ihre Grösse während der Lebensdauer verändern. Sie kann somit eine beliebige[1] Anzahl Elemente aufnehmen
  - Analogie: Luftballon
    - Je nach Gasvolumen deht sich der Luftballon räumlich aus oder zieht sich wieder zusammen

[1] natürlich begrenzt durch den verfügbaren Speicher

8.5 Explizite vs. implizite Beziehungen

- Bei **expliziten** Datenstrukturen werden die Beziehungen zwischen den Daten von jedem Element **selber explizit** mit Referenzen festgehalten
  - Analogie: Fahrradkette
    - Kettenglieder sind explizit miteinander verknüpft
    - jedes Kettenglied kennt seine Nachbarglieder
- Bei **impliziten** Datenstrukturen werden die Beziehungen zwischen den Daten **nicht** von den Elementen selber festgehalten
  - Die Beziehungen werden quasi von *aussen* definiert, z.B. über eine externe Nummerierung (Index)
  - Analogie: Buchregal mit Büchern
    - Bücher stehen einfach (ggf. auch geordnet) nebeneinander
    - das einzelne Buch weiss nicht wo es steht, bzw. hingehört

8.6 Aufwand von Operationen

- Der **Aufwand** (Rechnen- und Speicherkomplexität) variiert sowohl für die verschiedenen Operationen als auch (oft) in Abhängigkeit der enthaltenen Datenmenge in einer Datenstruktur
- Meistens interessiert uns *nur* die Ordnung, also wie sich der Aufwand in Abhängigkeit zur Anzahl der Elemente verhält
- Beispiele:
  - Buch auf einen Stapel legen (ungeordnet):
    $\mathcal{O}(1)$ →Konstant
  - Buch in der Bibliothek alphabetisch einordnen:
    im schlechtesten Fall $\mathcal{O}(n)$ →Linear
  - Eine unsortierte Menge von Büchern alphabetisch ordnen:
    im schlechtesten Fall $\mathcal{O}(n^2)$ →Quadratisch (Polynomial)

9 Arrays

9.1 Eigenschaften

- **Statische** Datenstruktur
  - Grösse wird bei Initialisierung festgelegt. Beispiel:

```
char[] demo = new char[10];
```

- **Implizite** Datenstruktur
  - Die einzelnen Elemente haben keine Beziehung untereinander bzw. keine Referenzen aufeinander
- **Direkter** Zugriff
  - Auf jedes Element kann über den Index direkt zugegriffen werden
- **Reihenfolge** der Elemente
  - Der Array behält die Positionen der Datenelemente, so wie sie zugewiesen / eingeordnet wurden, unverändert bei

Teil IV
SW 03 - Bäume

10 Lernziele

- Sie wissen wie eine baumartige Datenstruktur aufgebaut ist
- Sie kennen verschiedene Beispiele von Baumstrukturen
- Sie kennen die Grundelemente eines Baumes:
  Wurzel, Knoten, Blatt und Kanten
- Sie können die Kenngrössen eines Baumes beschreiben

10.1 Verwendung und Arten von Baumstrukturen

Zwei grundlegende Szenarien:

1. Die Daten haben bereits inhärent eine hierarchische Struktur, welche man entsprechend abbilden will. Beispiele:
   - Dateisystem mit Verzeichnissen und Dateien
   - Stammbaum (Genealogie)
   - Vererbungshierarchie in Java (nur mit Einfachvererbung)
2. Wenn man in einer geordneten Datenmenge einzelne Elemente sehr schnell finden will →binärer Suchbaum
   - Die Suche über eine Baumstruktur hat typisch nur einen Aufwand von $\mathcal{O}(\log n)$, und ist somit der rein sequenziellen Suche mit $\mathcal{O}(n)$ deutlich überlegen

- Mit Ausnahme der Wurzel (Ursprung des Baumes, die **alle** baumartigen Strukturen haben) können Bäume sehr stark variieren:
  - Unterschiedliche Anzahl Äste
  - Unterschiedliche Länge (Tiefe) der Äste
  - Die Breite (Grad) und die Höhe der Bäume ist sehr variabel
- Je nach Anwendungszweck definiert man mehr oder weniger **Restriktionen**, welche dann zu spezifischeren Baumstrukturen führen, welche auch spezifischere Eigenschaften aufweisen
  - Zweks Beschleunigung und/oder einfacherer Algorithmen

10.2 Gerichtete und ungerichtete Bäume

- Ein ungerichteter Baum ist eine reine Hierarchie
- Out-Tree, Navigation von der Wurzel **nach unten** zu den Blättern
  - →Kanten (Pfeile) gehen von der Wurzel aus. Am Häufigsten!
- In-Tree, Navigation von den Blättern **nach oben** zur Wurzel
  - →Kanten (Pfeile) zeigen zur Wurzel hin. Seltener.
- Diverse Spezialformen von Bäumen (Beispiele)
  - **Binär**-Baum - am einfachsten und häufigsten
  - **AVL**-Baum - höhenbalancierter Binärbaum
  - **B**-Baum - balancierter Baum, **nicht** zwingend binär!
  - **B***-Baum - restriktivere Form B-Baumes (ebenfalls balanciert)
  - **Binomial**-Baum - speziell strukturierter Baum
  - etc.

10.3 Kenngrössen von Bäumen

10.3.1 Ordnung

- Die **Ordnung** (order) eines Baumes definiert, wie viele Kinder ein Knoten **maximal** haben darf
  - Die Anzahl muss in eine konkreten (Teil-)BAum aber **nicht** zwingend erreicht werden
- Die Ordnung ist eine Definition!

10.3.2 Grad

- Der **Grad** (degree) eines Knotens sagt, wie viele Kinder ein bestimmter Knoten **aktuell** tatsächlich hat
- Bei einem Baum, z.B. der **fünften** Ordnung, darf der Grad jedes einzelnen Knotens **maximal 5** betragen, also maximal fünf Kinder

10.3.3 Pfad

- Als **Pfad** (path) eines Knotens bezeichnet man den Weg von der Wurzel bis zum entsprechenden Knoten, bzw. Blatt

10.3.4 Tiefe

- Die **Tiefe** (depth) eines Knotens beschreibt die Länge seines Pfades. Dazu werden die Kanten auf seinem Pfad gezählt
- Achtung: Es gibt auch eine Zählweise die bei 1 beginnt; es ist nicht einheitlich geregelt

10.3.5 Niveaus / Ebenen (levels)

- Als **Niveau** oder **Ebene** bezeichnet man die Menge aller Knoten, welche die gleiche **Tiefe** haben

10.3.6 Höhe

- Die **Höhe** (height) eines Baumes definiert sich aus der **Tiefe** des Knotens, welcher am **weitesten** von der Wurzel entfernt ist, bzw. aus der Anzahl der **Niveaus**

10.3.7 Gewicht

- Das **Gewicht** (weight) eines Baumes definiert sich über die Anzahl der enthaltenen Knoten

10.4 Füllgrade

10.4.1 Ausgefüllt

- Ein Baum wird als **ausgefüllt** bezeichnet, wenn **jeder innere Knoten** die **maximale** Anzahl an Kindern hat
- Der **Grad aller** inneren Knoten ist somit **gleich** der **Ordnung** des Baumes

10.4.2 Voll

- Ein Baum wird als **voll** bezeichnet, wenn das **letzte** Niveau linksbündig (oder auch rechts) angeordnet ist, und alle **restlichen** Niveaus die **maximale** Anzahl an Kindern enthalten

10.4.3 Vollständig oder komplett

- Ein Baum wird als **vollständig** oder **komplett** bezeichnet, wenn **jedes** Niveau die **maximale** Anzahl Knoten enthält
  - Er hat dann für sein **Gewicht** die **minimale** Anzahl **Niveaus**
  - Die Struktur ist immer **symmetrisch** und ausgeglichen

11 Binäre Bäume

11.1 Lernziele

- Sie sind mit binären Bäumen vertraut
- Sie kennen die Algorithmen, um binäre Bäume auf unterschiedliche Arten zu traversieren
- Sie sind mit den spezielen Eigenschaften von binären Suchbäumen vertraut
- Sie wissen wie das Suche, Einfügen und Entfernen von Knoten in binären Suchbäumen konzeptionell abläuft
- Sie verstehen, was ein ausgeglichener Baum ist und wie man diesen Zustand herstellen kann

11.2 Binärer Baum

- Ein **binärer Baum** (binary tree) ist als Baum mit **Ordnung 2** definiert. Jeder Knoten hat somit maximal **zwei** Kinder
  - Diese werden als **linker und rechter** Kindknoten bezeichnet
- Binäre Bäume sind in der Informatik **sehr** beliebt, weil:
  - durch die Beschränkung auf die **Ordnung 2** einige Algorithmen stark vereinfacht werden
  - auf binären Bäumen unterschiedliche Durchlaufordnungen (→Traversierungen) möglich sind
  - die Suche in einem binären (Such-)Baum einer binären Suche entspricht

11.3 Traversieren eines binären Baumes

- Aufgrund der spezifischen Eigenschaft von binären Bäumen (Ordnung 2) kann man diese auf **drei** unterschiedliche Arten traversieren (vergleiche dazu Iteration bei **Listen**)
  - **Preorder** - Hauptreihenfolge
  - **Postorder** - Nebenreihenfolge
  - **Inorder** - Symmetrische Reihenfolge
- Die Algorithmen, welche diese drei verschiedenen Traversierungsarten beschreiben, sind alle **rekursive** Algorithen
- Alle Traversierungen sind direkt abhängig von der Anzahl Knoten und haben somit einen Aufwand von $\mathcal{O}(n)$

11.3.1 Preorder

11.3.2 Postorder

11.3.3 Inorder

11.4 Binäre Suchbäume

11.5 Geordneter binärer Suchbaum