Informatik II
-
Algorithmen und Datenstrukturen
-
Betriebssysteme

Markus Alpers
B.Sc. Media Systems und Ausbilder f. Industriekaufleute

12. Juli 2016

Inhaltsverzeichnis

0.1 Darüber reden wir in diesem Kurs 3

I Arrays **5**

1 Eine erste Aufgabe: Highscore-Liste **6**
1.1 Vertiefung: Variablen in der Realität 7
1.1.1 Speicher im Computer 8
1.1.2 Datenübertragung 10
1.1.3 Prozessoren 10
1.1.4 Datentypen und Computerspeicher 11
1.1.5 Vom Datentyp zur Datenstruktur 12
1.1.6 Zusammenfassung und Abschluss 12
1.2 Highscore-Liste und Array 13

2 Einfache Sortieralgorithmen **16**
2.1 Insertion Sort 16
2.2 Die Landau- bzw. O-Notation 18
2.2.1 Der Bezug zur Realität 20
2.2.2 Programmierung eines Insertion Sort 23
2.2.3 Komplexität der Laufzeit eines Algorithmus 23
2.2.4 Komplexität des Speicherbedarfs von Insertion Sort . 24
2.3 Speicherkomplexität S(n) 25
2.4 Bubble Sort 27
2.4.1 Noch mehr Bezug zur Realität 29
2.4.2 Sortieren mit Parallelprozessoren 29
2.5 Programmierung von Bubble Sort 30
2.6 Zusammenfassung zu einfachen Algorithmen 32

3 Laufzeit-effiziente Algorithmen **33**
3.1 Merge Sort 33

II Verkettete Listen **34**

III Bäume **35**

IV Betriebssysteme **36**

Stichwortverzeichnis **37**

0.1 Darüber reden wir in diesem Kurs

Wahrscheinlich haben Sie bereits den Begriff des Algorithmus an der ein oder anderen Stelle in Bezug auf Programme gehört. Auch mit Datenstrukturen wie Arrays haben Sie schon zu tun gehabt, wenn Sie ein wenig programmiert haben. Deshalb lautet eine der häufigsten Fragen zu Beginn eines Kurses über Algorithmen und Datenstrukturen: „Was gibt's denn da noch zu lernen? In Java gibt es doch haufenweise Datenstrukturen, die können wir doch nutzen. Und Algorithmen sind doch auch klar...“

Diese Frage offenbart aber ein klares Missverständnis: **Datenstrukturen**, die wir in Programmiersprachen angeboten bekommen, erfüllen jeweils einen bestimmten Bedarf, den viele Programmierer haben. Sie erfüllen aber eben längst nicht jeden Bedarf und wenn wir gute Software entwickeln wollen, dann müssen wir im Stande sein, selbst eine Datenstruktur zu programmieren, die für ein neues Problem gut geeignet ist. Und umgekehrt müssen wir im Stande sein, zu erkennen, ob eine Datenstruktur, die in einer Programmiersprache enthalten ist schon alles anbietet, was wir für eine Aufgabe brauchen. Schließlich wäre es nicht sehr intelligent, etwas selbst zu programmieren, dass es bereits genau so gibt.

Das führt dann gleich zum nächsten Missverständnis: Informatik ist eben nicht Programmierung, sondern Informatik ist die Wissenschaft, in der wir nach Methoden suchen, um schwierige Probleme zu lösen. Wenn die Probleme einfach sind, kann sie ja jeder lösen, also schauen wir uns die schweren Probleme an. Und wenn es schon eine Lösung gibt, dann wäre es Zeitverschwendung etwas eigenes zu entwickeln, das nicht auch besser ist. Der Lösungsweg, den wir dabei für ein Problem finden wird als Algorithmus bezeichnet. Und damit sind wir bei dem Grund, aus dem wir uns mit Datenstrukturen und Algorithmen beschäftigen: **Algorithmen** sind Beschreibungen einer Problemlösung und Datenstrukturen beschreiben, wie wir die dabei verwendeten Daten systematisch aufbewahren. Nur gemeinsam machen die beiden Sinn.

Um das explizit zu ergänzen: Programmierung ist dann die Umsetzung von Algorithmen und Datenstrukturen in eine Programmiersprache. Wer also Programmieren kann, aber kein Informatiker ist, der kann zwar einen Computer dazu bringen, etwas zu tun, ist aber oft nicht im Stande zu entscheiden, wie brauchbar die programmierte Lösung für unterschiedliche Fälle ist. Schlimmer noch: Reine Programmiere versuchen teilweise Lösungen zu programmieren, bei denen InformatikerInnen nach einer kurzen Prüfung feststellen, dass eben diese Lösung nicht funktionieren kann. (Z.B. weil sie in 9 von 10 Fällen bis zum Ende des Universums dauert.) Viele reine ProgrammiererInnen neigen außerdem dazu, entweder eine Speziallösung zu

programmieren, die in den meisten Fällen gänzlich sinnlos ist oder sie entwickeln eine derart umfangreiche Lösung, dass damit auch lauter Aufgaben gelöst werden können, die niemals vorkommen werden. Letzteres ist allerdings auch ein typischer Fehler von InformatikerInnen.

Damit kommen wir zu einem dritten Begriff, den alle InformatikerInnen beherrschen: Wenn wir über **Komplexität** reden, dann meinen wir damit in aller Regel einen Maßstab, mit dem wir die Qualität verschiedener Algorithmen vergleichen können. Der bekannteste Maßstab für Komplexität ist die sogenannte O-Notation. Sie ist ein Maßstab, mit dem wir vergleichen können, wie viel Zeit verschiedene Algorithmen benötigen, die eine unbekannte Menge an gleichartigen Daten bearbeiten. Aber wir können uns auch die Komplexität des Speicherbedarfs verschiedener Algorithmen ansehen. Das ist schon alleine deshalb interessant, weil wir natürlich sicher stellen müssen, dass unser Programm nicht mehr Speicher belegt, als im Computer vorhanden ist.

Einer der wichtigsten Bereiche, in denen wir mit Algorithmen und Datenstrukturen zu tun haben sind die sogenannten Betriebssysteme. Am Ende dieses Kurses werden wir uns deshalb ansehen, was Betriebssysteme tatsächlich sind. All diejenigen von Ihnen, die bei Betriebssystemen sofort an Windows, iOS oder Linux denken werden sich hier umstellen müssen: Wie an vielen anderen Stellen im Studium reden wir hier nicht über die Konfiguration oder Nutzung eines Betriebssystems, sondern darüber, was die EntwicklerInnen eines Betriebssystems beachten müssen, bzw. darüber, was ein Betriebssystem leistet.

Die Inhalte dieses Kurses können Sie in den unterschiedlichsten Bereichen der Softwareentwicklung aber auch in der Administration nutzen. Denn diejenigen von Ihnen, die die Inhalte dieses Kurses gut verinnerlichen, werden später im Stande sein, mit nur wenig Aufwand verhältnismäßig schnell Lösungen zu entwickeln, die möglichst gut geeignet sind, um eine Aufgabe durch eine Computer oder ein Computernetzwerk zu lösen. Aber es geht noch darüber hinaus: Auch logistische Prozesse (z.B. eine bedarfsgerechte Planung von Verkehrsmitteln im Öffentlichen Personennahverkehr) basieren auf den Grundlagen von Algorithmen und Datenstrukturen.

Weiterführende Kurse laufen unter Namen wie Algorithmendesign, Algorithmik oder auch Graphentheorie.

Teil I

Arrays

Kapitel 1

Eine erste Aufgabe: Highscore-Liste

Stellen Sie sich vor, Sie wollen in einem Ihrer Programme eine Highscore-Liste programmieren, also eine Aufstellung der SpielerInnen, die ein Spiel gespielt haben und die nach dem erreichten Punktestand sortiert wird. Dann ist die wichtigste Frage nicht die, welchen Datentyp Sie für den Namen der einzelnen SpielerInnen wählen oder welchen Datentyp Sie für die Punkte wählen. Vielmehr ist die wichtigste Frage die, wie Sie die Sortierung so organisieren können, dass die Liste auch dann noch vollständig richtig sortiert ist, wenn in jeder Sekunde neue Einträge hinzukommen bzw. alte Einträge entfallen.

Sie fragen, warum da alte Einträge entfallen müssen? Ganz einfach: Stellen Sie sich vor, Spieler beenden ein Spiel alle drei oder fünf Minuten. Wenn nun nicht jeweils nur ein Ergebnis pro SpielerIn in der Liste auftaucht, eben das jeweils beste aller Spieldurchgänge, dann ist es kaum möglich, dass Spieler sich vergleichen. Es wird stets die Frage im Raum stehen, ob die SpielerInnen in der Liste doch noch irgendwo ein besseres Ergebnis geschafft haben. Und damit ist die Liste letztlich überflüssig.

Wenn Sie sich fragen, warum dieser Punkt so ausführlich besprochen wird, dann machen Sie sich bitte bewusst, dass Sie bei der Programmierung eben auch dieses Löschen direkt oder indirekt realisieren müssen; der Computer vergisst nicht einfach so, sondern dann und nur dann, wenn wir durch einen entsprechenden Programmteil sicherstellen, dass Daten entfernt werden.

Eine zweite wichtige Frage ist die, wie Sie den Teil der Liste anzeigen können, der jeweils für einzelne SpielerInnen interessant ist.

Reine Programmierer werden diese beiden Frage ignorieren und sich erst dann mit Ihr beschäftigen, wenn das Kind in den Brunnen gefallen ist, also dann, wenn die Spieler sich reihenweise im Forum zum Spiel darüber beklagen, dass die Highscoreliste „voll buggy“ oder „zu langsam“ ist. Und tatsächlich ist die Entwicklung einer „guten“ Highscore-Liste eine ausgesprochen anspruchsvolle Aufgabe. Schauen wir uns also an, wie wir mit diesem Problem umgehen können, bzw. was es bedeutet, wenn wir die Sortierung auf unterschiedliche Weise angehen.

Anm.: Ein Ziel dieses Kurses besteht darin, dass Sie einige Standard-Algorithmen kennen, und dass Sie im Stande sind, zu bewerten, wie gut diese Algorithmen für ein bestimmtes Problem geeignet sind. Um das etwas anschaulicher zu gestalten werden Sie diese Algorithmen bzw. jeweils passende Datenstrukturen in Java programmieren.

1.1 Vertiefung: Variablen in der Realität

Die Überschrift mag etwas seltsam klingen: Sie haben Variablen bislang als etwas kennen gelernt, dass in einer Programmiersprache dazu dient, um Werte zu speichern. Also stellt sich nun die Frage, warum es hier einen Abschnitt zu Variablen „in der Realität“ geben soll.

In diesem Abschnitt behandeln wir mehrere Aspekte, die in den Bereich der **Technischen Informatik** gehören, da wir tatsächlich über die technischen Grundlagen von Computern reden, die für unseren Kurs relevant sind.

Tatsächlich ist das, was Sie als Variablen in einer Programmiersprache wie Java oder PHP kennen gelernt haben aber nur eine stark vereinfachte Sicht auf das, was tatsächlich im Computer passiert. Und ohne ein besseres Verständnis dafür, was dort tatsächlich passiert werden Sie nicht im Stande sein, Datenstrukturen zu verstehen. Damit wiederum werden Sie nicht im Stande sein, umfassend zu verstehen, warum die Beschäftigung mit Algorithmen und Datenstrukturen so essenziell ist.

Schauen wir uns dazu zunächst an, woraus ein Computer besteht. In anderen Worten: Schauen wir uns die **Computerarchitektur** an. Im Folgenden geht es nicht um die Konzepte des von-Neumann-Rechners oder der Harvard-Architektur, die sonst einen Einstieg in dieses Thema bietet. Vielmehr schauen wir uns Computerarchitektur vor dem Hintergrund an, wie Variablen dort tatsächlich vorliegen und was alles nötig ist, damit wir Sie so nutzen können, wie das in Programmiersprachen der Fall ist.

Grunsätzlich haben wir bei Computern (mindestens) einen Prozessor, (mindestens) einen Speicher und (mindestens) einen Weg, um Daten zwischen den beiden zu transportieren. Um es uns leicht zu machen werden wir im folgenden so tun, als wenn es von all diesen Dingen jeweils nur eines gäbe.

Der Speicher und der Prozessor sind bei heutigen Computern einzig im Stande, Binärwerte, also die zwei unterschiedliche Symbole zu verarbeiten. Die bezeichnen wir in der Informatik als 1 und 0. Was diese 1 und 0 in der realen Welt alles sein können ist Thema von Veranstaltungen zur Nachrichten- und Kommunikationstechnik. Wie in der Informatik üblich vereinfachen wir die Situation, indem wir so tun, als wenn es die 1 und die 0 als Zahl gäbe.

Als NutzerInnen von Computern sind wir es dagegen gewöhnt, dass Computer uns Bilder und Texte anzeigen sowie Musik spielen. Außerdem können wir mit ihnen diese drei Medien übertragen. Weiterhin können wir mit den unterschiedlichsten Geräte Computer bedienen. Da wären Touchscreens, Mäuse, Tastaturen, usw. In Programmieren 1 haben Sie gelernt, dass all das bereits durch die Benutzung von Variablen erreicht werden kann.

Wir müssen uns jetzt also ansehen, wie es möglich ist, von Einsen und Nullen zu Variablen zu kommen, um zu verstehen, was eine Programmiersprache leisten muss. Zum Teil werden diese Aufgaben von Betriebssystemen übernommen, zum Teil von Programmiersprachen. Die Grenze ist dabei nicht festgelegt und darum unterscheiden wir an dieser Stelle nicht, wo die Aufgaben gelöst werden.

1.1.1 Speicher im Computer

Damit wir Variablen nutzen können ist die erste Voraussetzung, dass wir **addressierbaren Speicher** nutzen können. Wie Sie wissen befindet sich in jedem Computer Speicher. Diesen können Sie sich vereinfacht als eine nahezu endlose Fläche mit Karopapier vorstellen: Jedes Kästchen enthält eine 1 oder eine 0. Wenn wir damit arbeiten wollen, dann müssen wir im Stande sein, jedes Kästchen bewusst anzusprechen. Denn sonst ist der Speicher für uns wie ein Telefon: Wenn wir keine Möglichkeit haben, andere Personen bewusst auszuwählen, dann nützt uns das Telefon nichts. Beim Telefonieren nutzen wir dafür Telefonnummern: Zahlen, denen jeweils genau ein bestimmter Anschluss zugeordnet ist.

Genau dasselbe brauchen wir auch beim Computerspeicher, nur dass wir hier nicht von Telefonnummern, sondern von **Addressen** reden. Und genau wie eine Telefonnummer jedem Anschluss zugeordnet werden muss, muss auch beim Speicher eines Computers jedem Bereich eine Addresse

zugeordnet werden. Das ist eine der Aufgaben, die nie von der Programmiersprache, sondern immer vom Betriebssytem übernommen wird.

Bei vielen Rechnern wird der Speicher in Einheiten von 8 Bit Länge einzuteilen. Bei vielen, aber längst nicht bei allen, also seien Sie auf Systeme mit einer anderen Speichereinteilung vorbereitet.

Das bedeutet, dass jede Adresse auf einen Speicherbereich verweist, in dem wir eine Zahl von 0 bis 255 speichern können. Anders ausgedrückt (und das ist wichtig), können wir mit einem solchen Speicherbereich 256 verschiedene Symbole „speichern". Wir können in solch einem Speicherbereich auch die Belegung von 8 Leitungen speichern, über die unser Computer Daten empfängt oder sendet.

Diese beiden unterschiedlichen Interpretationen dienen dazu, dass Sie verstehen, dass ein 8-Bit-Speicher nicht ausschließlich dafür genutzt werden kann, um die Zahlen von 0 bis 256 speichern zu können; denn wofür die einzelnen Ziffern und Zahlen stehen, darüber sagt der gespeicherte Wert nichts aus. Wichtig ist auch, dass Sie sich vergegenwärtigen, dass diese Einteilung des Speichers in 8-Bit-Einheiten eine willkürliche Festlegung ist und dass es möglich ist, mehr oder weniger Bit pro Speicheradresse zusammenfassen. Es ist dabei weitgehend irrelevant, was für ein Prozessor im Computer steckt.

Im Gegensatz dazu hat Speicher immer auch eine Zugriffsgeschwindigkeit, die durch seine Bauweise und Spezifikation festgelegt ist. Hierauf können wir im Regelfall nicht zugreifen, sondern müssen mit der Geschwindigkeit zurechtkommen, für die der Speicher konfiguriert ist. Eine höhere Geschwindigkeit als die Lichtgeschwindigkeit ist aber nicht möglich. Als grobes Augenmaß können Sie sagen, dass die 4 GHz, also $4 \cdot 10^9$ Operationen pro Sekunde, die ein Prozessor ausführen kann die Obergrenze für jede Art der Datenübertragung bildet. Speicher in aktuellen Desktoprechnern arbeitet üblicherweise mit weniger als 300 MHz, also werden maximal $3 \cdot 10^8$-mal pro Sekunde Daten zwischen Speicher und Prozessor ausgetauscht.

Einige von Ihnen werden jetzt vielleicht protestieren, weil in Ihrem Computer Speicher eingebaut ist, der mit 3 GHz arbeitet, aber das basiert auf einem Missverständnis: Die tatsächliche Taktung von Speicher kann z.B. 200 MHz sein. Daraus ergeben sich „3 GHz" über einen Rechentrick: Vereinfacht ausgedrückt werden bei jeder Datenübertragung 15 Byte übertragen, anstelle von nur einem. $15 \cdot 200$ *MHz = 3 GHZ. Das sind aber eben keine realen 3 GHz und es ist auch nicht der volle Geschwindigkeitsgewinn, weil eben nicht 15 Byte übertragen werden, die für den Programmablauf benötigt werden, sondern es wird in diesem Fall ein Byte übertragen, das benötigt wird und 14 Byte, die im Speicher darauf folgen. Es ist*

richtig, dass benachbarter Speicher häufig gemeinsam benutzt wird und deshalb ergibt ein solcher Multiplikator auch durchaus Sinn: Dieser Multiplikator erhöht tatsächlich die Menge der übertragenen Daten pro Sekunde, aber nicht alle dieser Daten werden tatsächlich benötigt. Viel wichtiger ist hier also der Takt, mit dem der Speicher arbeitet, denn das erste Byte, das in einem Takt übertragen wird wird auch immer benötigt.

Zusammenfassung:

Sie haben jetzt ein Grundverständnis dafür, was Computerspeicher ist: Eine geordnete Ansammlung von direkt ansprechbaren Bereichen, in denen wir Binärwerte einer gewissen Größe speichern können.

1.1.2 Datenübertragung

Aus der Sicht der Informatik sind Wege zur Datenübertragung gewissermaßen eine Black Box, von der wir nur wissen, dass wir in einem bestimmten Zeitraum eine bestimmte Menge an Daten maximal darüber übertragen können. Die Nachrichten- und Kommunikationstechnik ist dann der wissenschaftliche Bereich, in dem die Techniken und Technologien entwickelt werden, die uns die Datenübertragung überhaupt ermöglicht. Sie untersucht auch, wie und wie gut Fehler bei der Datenübertragung automatisch korrigiert oder zumindest erkannt werden können.

Für uns ist in diesem Kurs einzig die resultierende Geschwindigkeit relevant. Sie werden allerdings feststellen, dass wir auch diese nicht konkret untersuchen, da wir uns auf die Geschwindigkeitsunterschiede von unterschiedlichen Algorithmen konzentrieren werden, die die gleiche Aufgabe erfüllen. Und das das Verhältnis der beiden zueinander nicht von der Geschwindigkeit der Datenübertragung auf unterschiedlichen Systemen abhängt, ist eben diese Geschwindigkeit im Rahmen dieses Kurses irrelevant.

1.1.3 Prozessoren

Damit kommen wir zu dem Teil eines Computers, der die eigentliche Verarbeitung leistet. Ähnlich wie die Datenübertragung spielt der Prozessor selbst in diesem Kurs nur eine kleine Rolle: Bezüglich der Geschwindigkeit gilt dasselbe wie für die Datenübertragung. Allerdings werden wir gelegentlich auf Parallelprozessoren eingehen.

Der Vollständigkeit halber sei hier noch angemerkt, was es mit Prozessoren wie einem 64-Bit-Prozessor auf sich hat: Die **Bittigkeit** eines Prozessors (in

diesem Fall 64) sagt aus, wie viele Bit der Prozessor gleichzeitig verarbeiten kann. Sie gibt damit auch an, wie viel Speicher ein Prozessor maximal verwalten kann: Ein 8-Bit-Prozessor kann maximal 2^8, also 256 Bereiche im Speicher addressieren, weil er eben nur bis zu 256 Zahlen kennt. Das ist auch der Hauptgrund, aus dem vor einigen Jahren 64-Bit-Prozessoren auf den Markt kamen: Ein 32-Bit-Prozessor kann maximal 4 Mrd. Einheiten Speicher verwalten (bei einem Byte pro Einheit ergibt das die Obergrenze von 4 GB).

Rechnen Sie es ruhig nach: Wir haben also mit 64-Bit-Prozessoren die Möglichkeit, mehr Speicherbereiche direkt anzusprechen als das Universum Atome hat. Damit können heutige Prozessoren genug Speicher nutzen, um jedes lösbare Problem tatsächlich zu bearbeiten.

1.1.4 Datentypen und Computerspeicher

Aus Sprachen wie Java kennen Sie statische Datentypen. Bislang wissen Sie lediglich, dass es sich dabei um eine Festlegung handelt, was für eine Art von Wert in einer Variablen gespeichert werden kann. So haben Sie den Datentyp int kennen gelernt, mit dem Sie Zahlen speichern können, die zwischen 0 und $2^{32} - 1$ Bit groß sind. Mit dem Wissen der vorigen Abschnitte können Sie jetzt schlussfolgern, dass diese Zahlen (vermutlich) 4 Einheiten Speicher belegen. Hier hätten wir also einen Datentyp, der jeder Zahl zur Basis 10 einen Gegenwert als 2er Komplement im Speicher zuordnet.

Und genau das ist es, was **Datentypen** tun: Sie **legen fest, wie ein Binärwert im Speicher interpretiert wird.**

Schauen wir uns das einmal für char-Variablen an, also für Variablen an, die z.B. einen Buchstaben speichern können. Zwar können wir willkürlich eine Zuordnung (Interpretation) eines Binärwerts im Speicher zu einem Buchstaben festlegen, aber das wäre problematisch, weil diese Zuordnung eben unsere willkürliche Festlegung wäre, anstatt das wir einen international gültigen Standard hätten. Wir hätten somit eine (wenn auch sehr schwache) Verschlüsselung erzeugt. Wenn Sie wissen wollen, warum diese Verschlüsselung sehr schwach ist, dann freuen Sie sich schon einmal auf die Veranstaltung **Kryptographie**, denn dort werden Sie die Antwort auf diese Frage kennen lernen.

Hier wie auch an anderen Stellen nutzen wir wieder Ergebnisse aus der Nachrichtentechnik: In der Nachrichtentechnik werden auch Standards entwickelt, wie Binärzahlen z.B. als Buchstaben interpretiert werden können. Wenn wir eine Tabelle haben, die festlegt, welche Binärzahl welchem Buch-

staben (oder anderen Symbol) entspricht, dann wird das als **Codierung** bezeichnet. Codierung umfasst allerdings noch wesentlich mehr. Aber darüber hören Sie etwas in der Veranstaltung **Kommunikationstechnik**.

Die beste Codierung, die Sie zurzeit wie Texte in Programmen verwenden können ist die sogenannte UTF-Codierung. Sie ist deshalb am besten geeignet, weil sie entwickelt wurde, damit die Symbole jeder Sprache der Welt eindeutig einem Binärwert zugeordnet sind. Wenn Sie also einen Datentyp verwenden, der UTF umsetzt, dann können Sie jedes Symbol jeder Sprache weltweit direkt in Ihrem Programm verwenden. Allerdings hat UTF gegenüber Codierungen wie ASCII einen Nachteil: Sie benötigt bis zu 32 Bit (4 Byte) pro Zeichen. Sollten Sie also einen Rechner mit wenigen KB Speicher nutzen, könnte es nötig sein, auf UTF zu verzichten.

Das bringt uns zum zweiten Detail bezüglich Datentypen in der Realität: **Ein Datentyp** legt nämlich nicht nur fest, wie ein Binärwert aus dem Speicher zu interpretieren ist, sondern er **legt gleichzeitig fest, wie viel Speicher für eine Variable zu reservieren ist**: Eine Programmiersprache, in der eine char-Variable die Codierung ASCII umsetzt reserviert für jede dieser Variablen 8 Bit Speicher. Eine Programmiersprache, in der eine char-Variable dagegen die Codierung UTF umsetzt, reserviert für jede dieser Variablen 32 Bit Speicher.

1.1.5 Vom Datentyp zur Datenstruktur

Und damit können Sie auch umfassend verstehen, was der Unterschied zwischen einem Datentyp und einer Datenstruktur ist: Ein Datentyp sagt immer aus, wie viel Speicher eine Variable belegt. Und dieser Speicherbedarf ist für alle Variablen eines Datentyps identisch. Also können Sie jetzt verstehen, warum ein String kein Datentyp ist.

Aufgabe:

Begründen Sie in eigenen Worten, warum ein String kein Datentyp ist.

1.1.6 Zusammenfassung und Abschluss

Beim Programmieren erzeugen, nutzen und ändern wir Variablen, die einen Datentyp und einen Wert haben.

In der Realität haben wir einen **Bezeichner** (den Namen, den wir der Variablen geben), also etwas, das bereits eine Datenstruktur vom Typ String ist.

Mit ein wenig Denkschmalz können Sie sich jetzt erklären, warum es Programmiersprachen gibt, bei denen nur die ersten acht Buchstaben eines Bezeichners tatsächlich verwendet werden können. Richtig gelesen: Die übrigen Zeichen des Bezeichners werden von diesen Sprachen ignoriert. Die Variablen abcdefgah und abcdefgahcde sind in solchen Sprachen also identisch.

Dieser Bezeichner verweist auf eine (!) **Adresse** im Speicher. Was Sie noch nicht wissen: Dieser Verweis, wird als **Pointer** bezeichnet. Es gibt Programmiersprachen, in denen Sie den Pointer ähnlich dem Wert der Variablen selbst programmieren können. Das wird als **Pointerarithmetik** bezeichnet.

Viele Informatikstudierende scheitern an der Pointerarithmetik in C, weil Sie nicht begreifen bzw. weil ihnen nicht gesagt wurde, das der Pointer schlicht die Adresse ist, auf die eine Variable verweist.

Der **Datentyp** legt wiederum zwei Dinge fest:

(1) Wie groß der Speicherbedarf der Variablen ist, also wie viele Einheiten Speicher für die Variable reserviert werden, auf die die Variable verweist. In anderen Worten: Gehören nachfolgende Adressen zur Variable und wenn ja, wie viele sind es.

(2) Wie der Binärwert zu interpretieren ist, der in dem Speicherbereich gespeichert ist, der der Variablen zugeordnet ist.

Es ist wichtig, dass Sie diese Vertiefung grundsätzlich verstanden haben, da sie die Grundlage für alles darstellt, was wir in diesem Kurs besprechen werden. Das beginnt bereits bei der Besprechung von Arrays, bzw. bei der Antwort auf die Frage, warum das, was in PHP als Array bezeichnet wird keine Datenstruktur namens Array ist.

1.2 Highscore-Liste und Array

Ein **Array** ist ein Bereich im Speicher, der in gleichgroße Teile unterteilt ist. Jedes dieser Teile ist genau so groß, dass eine Variable eines bestimmten Datentyps darin gespeichert werden kann.

Es gibt aber auch mehrdimensionale Arrays. Wenn wir z.B. von einem fünfdimensionalen Array sprechen, dann bedeutet das, dass jeder der genannten Teile so groß ist, dass fünf Variablen darin gespeichert werden können. Ob diese fünf Variablen alle den gleichen Datentyp haben müssen oder ob jede davon einen individuellen Datentypen haben darf, das hängt von der Programmiersprache ab. Aber grundsätzlich gilt, dass alle Variablen in ei-

ner Dimension eines Arrays den gleichen Datentyp haben müssen. Analog zum fünf-dimensionalen Array können Sie sich andere mehr-dimensionale Arrays veranschaulichen.

Dementsprechend können wir bei einer dynamisch typisierten Sprache wie PHP gar keine Datenstruktur wie ein Array haben. In **dynamisch typisierten Sprachen** werden wir aber dennoch häufig Datenstrukturen finden, die als Array bezeichnet werden. Der Grund ist simpel: Da die Programmierung dieser Datenstrukturen genauso durchgeführt wird, wie das bei einem Array der Fall ist, wirkt es für reine ProgrammiererInnen so, als wenn es sich tatsächlich um Arrays handeln würde. Im Hintergrund ist aber eine ganz andere Datenstruktur am Werk, mit den jeweiligen Vor- und Nachteilen.

Das ist ein weiterer Punkt, in dem sich InformatikerInnen und ProgrammiererInnen unterscheiden: InformatikerInnen lernen zuerst, was die Datenstruktur Array ist, während ProgrammiererInnen zuerst etwas kennen lernen, dass in einer Programmiersprache als Array bezeichnet wird. Da die „Arrays“ in unterschiedlichen Sprachen aber teilweise nichts mit der Datenstruktur Array gemein haben, führt der Einstieg von reinen ProgrammiererInnen häufig zu Problemen, wenn sie neue Programmiersprachen erlernen wollen: Kaum eine Einführung in eine Programmiersprache geht auf die Unterschiede zwischen den allgemein definierten Datenstrukturen und der konkreten Umsetzung in der jeweiligen Sprache ein. Wenn Sie beispielsweise Java gelernt haben, dann werden Sie sehr aufpassen müssen, wenn wir in diesem Kurs über Hash-basierte Datenstrukturen sprechen.

Um den Einstieg zu erleichtern werden wir in den nächsten Abschnitten so tun, als wenn es nur die Datenstruktur Array gäbe. Später lernen Sie dann Datenstrukturen kennen, die sich sehr von Arrays unterscheiden.

Aufgabe:

- Erklären Sie in eigenen Worten, warum es in dynamisch typisierten Sprachen kein „echtes“ Array geben kann.

  Tipp: Die Antwort steht nicht im vorigen Absatz. Um diese Aufgabe zu lösen müssen Sie den Unterschied zwischen statischer und dynamischer Typisierung verstanden haben und sich bewusst machen, was während des Ablaufs eines Programms im Speicher passieren kann.

Nehmen wir an, wir haben ein zweidimensionales Array, in dem die ersten Ergebnisse gespeichert, aber noch nicht sortiert sind:

Horst	100
Susi	260
Max	10
Knock Knock	500
Super Richie	2321
Queen Alice	91243
Zombie Maniac	2212
Cammy-Ka-Tse	90
Say cheeeeeeeeze	1203

Damit daraus eine wunschgemäße Highscore-Liste wird, müssen wir nun etwas entwickeln, das als **Sortieralgorithmus** bezeichnet wird.

Die Bezeichnung folgt dabei direkt aus der Aufgabe eines Algorithmus: Ein Sortieralgorithmus heißt Sortieralgorithmus, weil er ein Algorithmus ist, der die Elemente einer Menge (hier die Einträge eines Arrays) sortiert.

Und nun raten Sie mal, was ein Suchalgorithmus ist.

Kapitel 2

Einfache Sortieralgorithmen

Aufgabe:

- Notieren Sie eine Methode, die genau beschreibt, was zu tun ist, um eine solche Highscore-Liste zu sortieren. Die Reihenfolge soll dabei der Punktstand der SpielerInnen sein.

  Tipp: Wenn Sie hier nicht genau wissen, wie Sie vorgehen sollen, dann programmieren Sie das Ganze, wobei Sie die Liste als ein-dimensionales Array von Zahlen implementieren.

- Arbeiten Sie danach die folgenden Sortieralgorithmen durch und versuchen Sie sich insbesondere die Unterschiede bei der Laufzeit klar zu machen.

2.1 Insertion Sort

Wenn Sie häufig Karten spielen (oder gespielt haben), dann werden Sie die Einträge der Highscore-Liste vielleicht wie Karten behandeln, die Sie nacheinander auf ein Hand nehmen.

Der Sortieralgorithmus Insertion Sort basiert auf diesem Vorgehen, allerdings nehmen wir hier die unsortierten Karten direkt auf die Hand und beginnen dann von links nach rechts (oder umgekehrt) mit der Sortierung.

Bezüglich der Datenstruktur bedeutet das: Die Karten liegen als Elemente eines Arrays vor, Sie werden also nicht im Laufe des Algorithmus eingefügt, sondern lediglich umsortiert.

Anm.:

Hier und bei allen nachfolgenden Algorithmen werden wir uns auch gleich ansehen, wie viele Austausch- oder Vergleichsoperationen wir maximal brauchen, um eine gegebene Menge Karten zu sortieren.

Im folgenden Beispiel gilt, dass wir die Karten von klein nach groß von links nach rechts sortieren:

- Die erste Karte ist (verglichen mit sich selbst) bereits sortiert. Für das Vergleichen brauchen wir also keine Operation.
- Die zweite Karte vergleichen wir mit der ersten und vertauschen Sie dann oder lassen Sie an der selben Stelle. Danach sind die ersten beiden Karten sortiert. Wir haben bis hierher also 0 + 1 Vergleichsoperationen.

  Beim nächsten Schritt gibt es ein häufiges Missverständnis. Deshalb hier eine Klarstellung: Wenn wir von der ersten, der zweiten oder ... Karte sprechen, dann meinen wir in diesem Zusammenhang immer die Karte, die sich jetzt an erster, zweiter oder ... Stelle befindet. Das führt immer dann leicht zu Missverständnissen, wenn gerade eine Karte getauscht wurde, denn nun befindet sich eine andere Karte an der ... Stelle. Um präzise zu sein müsste die Beschreibung also jeweils lauten: Die Karte, die sich jetzt an ... Stelle befindet.
- Die dritte Karte vergleichen wir zuerst mit der zweiten. Wenn sie größer als die zweite ist, ist sie damit sortiert, denn die zweite Karte ist ja in der letzten Iteration bereits nach Größe sortiert worden. Ist die dritte Karte dagegen kleiner als die zweite, dann werden die beiden umgehend vertauscht. In diesem Fall folgt noch ein Vergleich und ggf. ein Austausch mit der ersten Karte. Für die dritte Karte brauchen wir also mindestens eine Vergleichsoperation aber maximal zwei. Im besten Fall haben wir also 0 + 1 + 1, im schlimmsten Fall 0 + 1 + 2 Vergleichsoperationen.

Aufgaben:

- Rechnen Sie den besten und schlimmsten Fall für fünf Karten durch.

  Den durchschnittlichen Fall und weitere Varianten lassen Sie vorerst außen vor. Diese haben wir nicht besprochen, auch wenn es sinnvolle Anwendungsgebiete dafür gibt.
- Formulieren Sie das Ergebnis als Summenformel.
- Formulieren Sie das Ergebnis jetzt für n Karten.

  Wenn Sie jetzt nicht an einen Beweis per Induktion denken, dann haben Sie zu wenig Zeit für Mathematik 1 investiert.

- Prüfen Sie nun, in welcher der folgenden Größenordnungen Ihr Ergebnis am ehesten liegt:
  - Eine konkrete ganze Zahl
  - log_2 n (in der Informatik meist als log n abgekürzt)
  - Die Variable n
  - n log_2 n (in der Informatik dementsprechend meist als n log n abgekürzt)
  - n^2
  - n^3
  - 2^n
  - n^n

Wenn Sie die vierte Aufgabe erfolgreich abgeschlossen haben, dann haben Sie eine Analyse der **Laufzeitkomplexität** durchgeführt.

Wichtig:

Für große Datenmengen ist es weniger wichtig, ob ein Algorithmus nun eine Komplexität von O(2 n^2) oder O(3 n^2) hat. Viel wichtiger ist die Antwort auf die Frage, ob sie bei n^2 oder n^3 oder doch bei **n log n** liegt. Deshalb musste Sie im vierten Aufgabenteil entscheiden, in welchem Bereich Ihr Ergebnis liegt. Wir reden deshalb auch von **Komplexitätsklassen**, bzw. von Algorithmen einer Komplexitätsklasse. Diese mögen sich dann im Detail unterscheiden, aber wir suchen weniger nach den Verbesserungen im Detail, als vielmehr nach der Antwort auf die Frage, ob ein Algorithmus für praktische Probleme nutzbar ist. Genau das ist der Grund, warum Bereiche wie „Algorithmen und Datenstrukturen“ zur Praktischen Informatik zählen: Wir suchen hier nach Lösungen für die Praxis, auch wenn die Mittel eher theoretisch wirken.

Diese Klassen unterscheiden sich allerdings von denen, die Sie in Veranstaltungen der Theoretischen Informatik kennen lernen werden: In der Theoretischen Informatik beschäftigen wir uns vorrangig mit Fragen wie derjenigen, ob ein bestimmte Klasse von Problemen überhaupt lösbar ist. Es geht dort also nicht um die praktische Relevanz einzelner Algorithmen, sondern um Grundlagenforschung.

2.2 Die Landau- bzw. O-Notation

Für den schlimmsten Fall nutzen wir die sogenannte O-Notation, die Teil der sogenannten Landau-Notation ist. Ausgesprochen wird es als Groß-O

Notation ausgesprochen, also nicht als Null-Notation. Sie können sich das so merken, dass das O für obere Grenze steht. Angenommen, Sie haben jetzt berechnet, dass die Laufzeitkomplexität im schlimmsten Fall 2^n beträgt, dann würden Sie das in der O-Notation als $O(2^n)$ notieren. Wobei die Laufzeitkomplexität für Insertion Sort NICHT $O(2^n)$ ist.

Ähnlich wie der dekadische Logarithmus ein Maßstab ist, mit dem wir Größenordnungen in der Natur beschreiben können, ist die Landau-Notation ein Maßstab, um die Größenordnung der Schwierigkeit einer Aufgabe bzw. eines Lösungsansatzes zu beschreiben. Anschaulich ließe sich diese Notation wie folgt zusammen fassen:

- Eine konstante Zahl drückt aus, dass es sich um eine einfache Aufgabe handelt.
- Ein Ausdruck, in dem die Schwierigkeit durch den Logarithmus einer Variablen ausgedrückt wird ist in aller Regel gut lösbar.
- Eine Variable (erster Potenz) drückt aus, dass die Aufgabe zwar umfangreich, aber in aller Regel zu bewältigen ist.
- Exponentielle Ausdrücke sind dagegen in aller Regel sehr anspruchsvoll oder sogar generell nicht zu bewältigen. In letzterem Fall spricht man auch von einem nicht-lösbaren Problem.

In der Praktischen Informatik vergleichen wir, welcher Algorithmus bzw. welche Algorithmen am besten geeignet sind, um eine Aufgabe zu lösen. Wenn wir die Landau-Notation auf die Laufzeit eines Algorithmus übertragen, dann ließen sich die verschiedenen Klassen wie folgt übersetzen:

- Ein Algorithmus, dessen Laufzeit nach O-Notation mit einer konstanten Zahl angegeben werden kann, wird immer nach der (nahezu) gleichen Zeit fertig sein, egal wie viele Daten er verarbeiten muss. Er ist also in zeitlicher Hinsicht äußerst effizient.
- Ein Algorithmus, dessen Laufzeit nach O-Notation $log_2 n$ oder n oder $n \cdot log_2 n$ beträgt, ist immer noch recht effizient. Tatsächlich gibt es viele häufig zu lösende Aufgaben, bei der eine höhere Effizienz als $n \cdot log_2 n$ nicht erreicht werden kann.

  Die Beweisverfahren, mit denen gezeigt wird, ob eine höhere Effizienz für eine bestimmte Aufgabe überhaupt möglich ist werden in Veranstaltungen der Theoretischen Informatik behandelt.
- Sobald wir es dagegen mit exponentiellen Ausdrücken in O-Notation zu tun bekommen, haben wir es mit Algorithmen zu tun, die in der Praxis nur in wenigen Fällen anwendbar sind. Hier wird zwischen

Algorithmen unterschieden, die in O-Notation als polynomieller[1] oder nicht-polynomieller Ausdruck angegeben werden.

Diese werden verkürzt als Algorithmen der Klassen **P** (kurz für „in polynomieller Zeit lösbar") und **NP** (kurz für „nicht in polynomieller Zeit lösbar") bezeichnet. Sie werden später (vorrangig in den Veranstaltungen der Theoretischen Informatik) auch Formulierungen wie „Das Problem ist **NP-hart**" lesen. Diese Formulierung besagt, dass die Aufgabe grundsätzlich nicht in polynomieller Zeit lösbar ist. Das bedeutet nichts anderes, als dass es (bis auf wenige Spezialfälle) unmöglich ist, ein Programm zu verfassen, dass diese Aufgabe mit einem Computer lösen kann.

2.2.1 Der Bezug zur Realität

Um es in Worten auszudrücken: Ein Algorithmus mit $O(2^n)$ benötigt also 2^n Operationen, um n Daten zu bearbeiten. Hier steht bearbeiten, denn die gleiche Art der Laufzeitanalyse können wir für jeden Algorithmus verwenden, der Datenmengen bearbeitet, egal ob er sie nun sortiert oder etwas anderes mit jedem einzelnen Datum macht.

Schauen wir uns das mal im Detail an: Ein Algorithmus mit $O(2^n)$ benötigt also im schlimmsten Fall rund 10^{33} Operationen, um 100 Daten zu sortieren. Nehmen wir an, wir können für unsere Aufgabe einen Prozessor mit einer Geschwindigkeit von 4 GHz nutzen, also einen Prozessor, der $4 \cdot 10^9$ Operationen pro Sekunde ausführen kann. Dann würde dieser Prozessor immer noch mehr als 10^{23} Sekunden für die Lösung dieses Problems benötigen.

Aufgabe:

- Angenommen, Sie hätten einen solchen Algorithmus programmiert. Berechnen Sie die Dauer, die dieser auf dem genannten Prozessor braucht, um seine Aufgabe abzuschließen. (Gemeint ist, dass Sie die Dauer in eine Form umrechnen sollen, die für Sie aussagekräftig ist... Schließlich ist die Angabe in Sekunden bei dieser Größenordnung für uns als Menschen eher schlecht einzuordnen.)

  Haben Sie eine realistische Chance, das er diese Aufgabe vor Ihrem Eintritt in die Rente abgeschlossen hat?

Aufgabe:

[1] Zur Erinnerung: Polynomieller Ausdruck: Ein Ausdruck, der sich als $\sum_{i=0}^{n} a_i \cdot x^i$ angeben lässt, also als $a_0 + a_1 \cdot x + a_2 \cdot x^2 + ... + a_n \cdot x^n$

- Berechnen Sie jetzt die vorige Aufgabe für einen Algorithmus durch, der die Aufgabe in O(n) lösen kann.

Aufgabe:

- Ergänzen Sie die folgende Aufstellung, um Zeiträume in Größenordnungen von Sekunden einzuordnen. Suchen Sie nach weiteren Größenordnungen für Zeiträume, die für Computer oder Menschen oder Tiere oder Unternehmen oder beliebige andere Dinge relevant sind.

  Beispiele:

  - 10^{-10} bis 10^{-9} Sekunden:
    Ausführungszeit für eine Operation auf Prozessoren, die im GHz-Bereich arbeiten.
  - 10^{0} Sekunden:
    Gewöhnliche Reaktionszeit eines Menschen.
  - 10^{1} bis 10^{2} Sekunden:
    Reaktionszeit eines angetrunkenen oder unter Drogen stehenden Menschen.
  - 10^{17} Sekunden:
    Ungefähre Dauer der bisherigen Existenz dieses Universums.

Aufgabe:

- Nehmen Sie nun die Tabelle, die Sie gerade erstellt haben und kategorisieren Sie die verschiedenen Einträge wie folgt:

  Tragen Sie aber jeweils die Dauer in Sekunden mit ein.

  Die kurzen Sätze nach den Kategorien sollen lediglich veranschaulichen, über welche Zeiträume wir hier reden. Und diese Aufgaben sollen vorrangig dazu dienen, dass Sie ein Gefühl für die verschiedenen Klassen der O-Notation bekommen.

  - **Kategorie 1:**
    Zu schnell für das menschliche Auge.
  - **Kategorie k:**
    Auf die Schnelle zu erledigen.
  - **Kategorie log n:**
    In der Zwischenzeit könnten wir uns einen Kaffe holen.

- **Kategorie n:**
  Lass uns doch in der Zwischenzeit mal Essen gehen; in der Kantine gibts heute Burger.
- **Kategorie n log n:**
  Das könnte heute vielleicht noch klappen. Eventuell dauert's aber auch bis Ende nächster Woche.
- **Kategorie n^2:**
  Ich bin nicht sicher, ob das noch vor Weihnachten nächsten Jahres klappt.
- **Kategorie 2^n:**
  Wir sollten lieber nicht nach Los Angeles fliegen. Ich habe gehört, dass die da irgendwann ein Erdbeben erwarten.
- **Kategorie n^n:**
  Also neulich ja, da habe ich was über parallele Universen gehört. Lass doch mal eine Forschungsreise in eines davon machen. Mal schaun, wann's klappt.

Fazit und Ergänzung:

Sie sollten jetzt verstanden haben, dass die O-Notation ein extrem wertvolles Mittel bei der Entwicklung von Software ist.

In Büchern, in denen die Komplexitätsanalyse von Algorithmen behandelt wird, wird ein Thema in aller Regel außen vor gelassen: Wie ändert sich die Laufzeitkomplexität, wenn wir Algorithmen auf Mehrkernprozessoren ausführen lassen?

Wie schon eingangs erläutert werden uns in diesem Kurs auch nicht umfangreich mit diesem Thema beschäftigen, aber beim nächsten Algorithmus werden Sie lernen, dass die O-Notation nichts darüber aussagt, ob ein Algorithmus auf einem Rechner mit mehreren Tausend Kernen noch die gleiche absolute Rechenzeit benötigt wie auf einem Rechner mit nur einem Kern.

Allerdings gibt es dafür einen guten Grund: Selbst wenn wir ein System mit mehreren Millionen Kernen zur Verfügung haben und wir einen Algorithmus nutzen, der diese Vielzahl an Kernen vollständig ausnutzt, selbst dann wird ein Algorithmus mit O(n^n) niemals 100 Daten innerhalb Ihrer Lebensdauer abarbeiten.

2.2.2 Programmierung eines Insertion Sort

Da Sie im ersten Semester die Programmierung von Java kennen gelernt haben, werden die Quellcodes in diesem Buch ebenfalls in Java angegeben.

Wie Sie hier sehen, sind für jeden einzelnen Austausch zweier Elemente drei Operationen nötig. Dieses Detail vernachlässigen wir bei der O-Notation, weil dieser Mehrbedarf für alle Sortieralgorithmen gilt. Wenn Sie später die verschiedenen Algorithmen miteinander vergleichen werden Sie genau das leicht nachvollziehen können.

```
// Das zu sortierende Array ist: int[] values

void insertionSort()
{
for (int i = 0; i < values.length; i++)
{
for (int j = values.length - i; j < values.length; j++)
{
if (values[i] > values[i+1])
{
switchValues(i,j);
} // fi (values[i] > values[i+1])
} // rof (int j = values.length - i; j < values.length; j++)
} // rof (int i = 0; i < values.length; i++)
} // insertionSort()

void switchValues(int i, int j){
int x = values[i];
values[i] = values[i+1];
values[i+1] = x;
}
```

Zur Erinnerung:

Wenn wir eine Highscore-Liste sortieren wollen, in die alle Spieler eines online Games eingetragen sind, dann reden wir nicht über 100 sondern eher über mehr als 100.000 Einträge. Hier sind also Laufzeiten von $O(2^n)$ in praktisch jedem Fall inakzeptabel.

2.2.3 Komplexität der Laufzeit eines Algorithmus

Die Komplexität der Laufzeit eines Algorithmus in O-Notation berechnen wir wie folgt:

- Jede Operation wird mit dem Wert 1 bewertet.
- Ein Rumpf erhält als Wert die Summe seiner Operationen erhöht um die Werte der Funktionsaufrufe. (Ggf. muss also zunächst der Wert eines Funktionsaufrufs berechnet werden, bevor ein Rumpf bewertet werden kann.)
- Eine Kontrollstruktur wird mit dem Wert bewertet, der sich für den Fall ergibt, der den höchsten Wert hat (also den, bei dem die meisten Operationen ausgeführt werden).
- Bei Iterationen und Rekursionen bestimmen wir zunächst, wie oft sie im schlimmsten Fall durchlaufen/aufgerufen werden. Mit diesem Wert multiplizieren wir den Wert des Rumpfes. Wenn die Anzahl Iterationen von der Anzahl zu bearbeitender Daten abhängt, dann setzen Sie für diese Anzahl die Variable n an.
- Die Bewertung von Funktionsaufrufen ergibt sich analog.
- Abschließend wird die errechnete Formel soweit reduziert, dass eine der Klassifikationen resultiert:
  - Nur der höchste Summand bleibt erhalten.
  - Bei der Multiplikation einer (oder mehrerer) Variablen mit einer Konstanten wird die Konstante gestrichen.

Wichtig:

Bei Sprachen, bei denen Funktionen nicht durch ein Klammernpaar hervorgehoben werden muss ist es leicht, die Laufzeitkomplexität falsch zu berechnen. Prüfen Sie deshalb in diesen Fällen lieber mehrfach, ob Sie auch tatsächlich jede Funktion vollständig erfasst haben.

Insbesondere müssen Sie sich bewusst machen, dass Funktionen einer Sprache selbst eine Laufzeitkomplexität besitzen; im Falle z.B. von proprietären Sprachen kann es durchaus sein, dass Sie keine genaue Laufzeitanalyse durchführen können.

Proprietäre Sprache bedeutet, dass diese Sprache rechtlich geschützt sind. Der Quellcode solcher Sprachen ist meist nicht öffentlich einsehbar und damit können Sie in diesen Fällen auch nicht prüfen, wie eine Funktion dieser Sprache arbeitet.

2.2.4 Komplexität des Speicherbedarfs von Insertion Sort

Wie eingangs erläutert, kann auch der Speicherbedarf eines Algorithmus ein wichtiges Kriterium sein. Schauen wir uns also die Implementierung

von Insertion Sort an, um etwas über den Speicherbedarf zu sagen. Wir haben hier leider keine explizite Notation wie die O-Notation für die Laufzeitkomplexität, aber das Prinzip ist dasselbe:

Zunächst haben wir das Array mit den gespeicherten Werten. Wenn wir wieder n als Variable für die Anzahl Elemente wählen und x als Variable für den Speicherbedarf in Byte, dann sind wir bei $x \cdot n$ Byte als Speicherbedarf.

Im besten Fall, also dann, wenn das Array bereits sortiert ist, ist das unser Speicherbedarf. Dann haben wir also keinen zusätzlichen Speicherbedarf durchs Sortieren.

Schauen wir uns also die Situation an, wenn sortiert werden muss. In diesem Fall wird zusätzlich Speicher für ein Element benötigt, weil dieses ja beim Vertauschen zwischengespeichert werden muss. Werfen Sie ggf. einen Blick in die Methode switchValues(), um das nachzuvollziehen. Damit haben wir $x \cdot n + x$ Byte oder $x \cdot (n + 1)$ als Komplexität des Speicherbedarfs im schlimmsten Fall.

In anderen Worten: Nur wenn die zu sortierenden Daten bereits dazu führen, dass der Speicher unseres Rechners vollständig gefüllt ist, nur dann ist die Sortierung mit Insertion Sort wegen des Speicherbedarfs nicht mehr möglich. Im Gegensatz zur Laufzeitkomplexität ist die Speicherkomplexität also proportional zur Menge der zu verarbeitenden Daten.

2.3 Speicherkomplexität S(n)

Ähnlich wie bei der Laufzeitkomplexität können wir jetzt also die Komplexität des Speicherbedarfs (kurz Speicherkomplexität) klassifizieren.

In anderen Kursen wird jedoch mit dem Begriff der Komplexität ausschließlich die Analyse der Laufzeitkomplexität verbunden. Deshalb sei hier betont, dass das S(n), das Sie gleich kennen lernen im Gegensatz zu O(n) kein allgemein anerkannter Maßstab ist. Wenn Sie also außerhalb dieses Textes auf die Abkürzung S(n) treffen, dann kann damit etwas anderes gemeint sein. Im Gegensatz dazu wird O(n) in der Informatik üblicherweise so verwendet, wie Sie es hier kennen gelernt haben.

Für diesen Kurs führen wir jetzt die Klassifizierung des Speicherkomplexität ein, die mit **S(n)** abgekürzt werden soll.

Ein Algorithmus der im schlimmsten Fall $17 \cdot n^2$ Bytes benötigt, um n By-

tes an Daten zu verarbeiten wird hier also mit S(n^2) klassifiziert. Analog zur O-Notation konzentrieren wir uns also auch bei der Klassifizierung der Speicherkomplexität ausschließlich auf den Bedarf in Abhängigkeit der zu verarbeitenden Datenmenge.

Allerdings gibt es bei der Berechnung von O(n) und S(n) gelegentlich Unterschiede: Wenn innerhalb eines Algorithmus zweimal nacheinander eine Schleife durchlaufen wird, die als Komplexität jeweils n hat, dann ergibt das eine Laufzeitkomplexität von 2 n. (Bitte nicht mit der abschließenden Klassifizierung verwechseln.) Wenn bei jeder diese Schleifen die Speicherkomplexität ebenfalls n ist, dann ist auch die gemeinsame Speicherkomplexität n.

Sie haben hier ein Verständnisproblem? Dann machen Sie sich bewusst, dass die abgelaufene Zeit beim ersten Schleifendurchlauf nicht wieder „verschwindet“, wenn der zweite Schleifendurchlauf beginnt. Im Gegensatz dazu wird der Speicher, der für den ersten Schleifendurchlauf benötigt wurde ja wieder freigegeben, bevor der zweite beginnt. Dieser Speicherbedarf erhöht also nicht zusätzlich zu der des zweiten Schleifendurchlaufs die Speicherkomplexität, sondern Sie müssen lediglich prüfen, welcher der beiden Schleifendurchläufe mehr Speicher benötigt. Und nur dieser Wert erhöht die Speicherkomplexität. Dagegen kann der zweite Schleifendurchlauf erst nach dem ersten beginnen, also erhöht sein Zeitbedarf die Laufzeitkomplexität des gesamten Algorithmus.

Das Gegenteil ist dann der Fall, wenn wir Teile eines Algorithmus parallel laufen lassen können, was vorrangig bei Rekursionen der Fall ist: Zwar benötigt jeder dieser Teile eine gewisse Zeit, aber da sie (nahezu) gleichzeitig ablaufen erhöht sich die Laufzeitkomplexität insgesamt nur um die Zeit, die derjenigen Teil benötigt, der am längsten zur Ausführung braucht. Dafür müssen wir in diesem Fall aber den Speicherbedarf addieren: Da alle Teile gleichzeitig Speicherbereiche belegen, addiert sich der Speicherbedarf.

Wenn Sie das verstanden haben, dann verstehen Sie auch, warum wir letzten Endes zwar ein Optimum aus Laufzeit- und Speicherbedarf erreichen können, es uns aber nahezu unmöglich ist, gleichzeitig eine hohe Effizienz im Speicher- und im Zeitbedarf zu erreichen.

An dieser Stelle sollte eines in aller Deutlichkeit festgehalten werden: Natürlich sollten Sie so viele Algorithmen und Datenstrukturen wie möglich auch nach diesem Kurs kennen und einsetzen können. Und ja, natürlich werden Sie auch darauf geprüft. Aber viel wichtiger ist, dass Sie den Inhalt des letzten Absatzes voll und ganz verinnerlichen: Dieses Verständnis unter-

scheidet InformatikerInnen von beliebigen ProgrammiererInnen.

2.4 Bubble Sort

Von nun an werden wir nicht mehr über Karten, sondern über Elemente einer Datenstruktur reden.

Bubble Sort ähnelt Insertion Sort, aber die Reihenfolge der verglichenen Elemente ist anders:

- Wir vergleichen paarweise Elemente:

  Das erste mit dem zweiten,
  aber nicht das zweite mit dem dritten,
  sondern erst wieder das dritte mit dem vierten,
  aber nicht das vierte mit dem fünften,
  sondern erst wieder das fünfte mit dem sechsten usw.
  und vertauschen jeweils, wenn nötig.

  (Wenn Sie das als Programmcode sehen, ist es viel übersichtlicher als diese Beschreibung, grübeln Sie also nicht zu sehr über die Logik.)

- Dann vergleichen wir wieder paarweise, aber dieses Mal:

  Nicht das erste mit dem zweiten,
  sondern direkt das zweite mit dem dritten,
  nicht das dritte mit dem vierten,
  sondern erst wieder das vierte mit dem fünften,
  nicht das vierte mit dem fünften,
  sondern erst wieder das sechste mit dem siebten usw.
  und vertauschen auch hier wieder, wenn nötig.

- Und so lange wir bei einem der beiden Durchläufe auch nur ein Element vertauscht haben, fangen wir anschließend wieder von vorne an.

Aufgabe:

- Berechnen Sie wie bei Insertion Sort die Laufzeitkomplexität von Bubble Sort.

Wenn Sie den Ablauf von Bubble Sort nachvollzogen haben, dann ist Ihnen wahrscheinlich intuitiv klar geworden, dass dieser eine ähnlich schlechte Laufzeit wie Insertion Sort haben muss. Und tatsächlich gehören beide bezüglich der Laufzeitkomplexität in eine Klasse.

Auch wenn Sie jetzt noch keinen effizienten Algorithmus in diesem Kurs kennen gelernt haben, sollte Ihnen eines klar geworden sein: Die Methoden, die wir in alltäglichen Problemen nutzen, um eine Aufgabe zu lösen sind für Aufgaben, für die wir Computer nutzen können meist mangelhaft bis ungenügend geeignet: Sie benötigen schlicht zu viel Zeit für die Massen an Daten, die damit verarbeitet werden müssen.

Das ist für die meisten Menschen verwunderlich, aber es ist nachvollziehbar: Die Aufgaben, die wir im Alltag erledigen haben nur selten mit derartigen Datenmengen zu tun, die Computer alltäglich bearbeiten.

Ein Beispiel, bei dem Sie die Ineffizienz im Alltag erleben können sind (Fußball-)Sammelbilder: Irgendwann hat jeder Sammler mehrere hundert Bildchen und hier zeigt sich, dass das Einsortieren sehr lange dauern kann, wenn wir keine effizienteren Methoden nutzen. Doch anstatt neue Ansätze auszuprobieren verschwenden die meisten Sammler lieber Stunden für das Einsortieren neuer Karten.

.

Aufgaben:

- Eine Sammlerin hat 3000 doppelte Sammelbilder und sortiert neue Doubletten naiv ein. Sprich, Sie beginnt an einem Ende Ihrer sortierten Bilder und vergleicht jedes mit einem neuen Bild, bis sie das neue Bild einsortieren kann. (Im Kern reden wir hier also über Insertion Sort, nur dass tatsächlich neue Elemente ins Array einsortiert werden.)

  Für fünf Vergleiche benötigt Sie eine Sekunde. Kalkulieren Sie mit der Laufzeitkomplexität von Insertion Sort: (a) Wie lange braucht Sie im schlimmsten Fall, um zwanzig neue Bilder einzusortieren. (b) Wie lange braucht Sie im besten Fall?

- Was kann sie tun, damit sie mit dem Einsortieren schneller fertig ist, ohne vom geschilderten Grundprinzip abzuweichen? (Gemeint ist, dass Sie am Algorithmus für das Einsortieren selbst nichts ändern darf.)

- Berechnen Sie den worst case (schlimmsten Fall) für die Laufzeit, wenn sie diese Verbesserung anwendet.

Anm.:

Keine Frage, niemand würde bei derartigen Mengen an Sammelbildern noch naiv sortieren, aber Sie werden im Laufe dieses Kurses erkennen, wie

extrem die Zeitgewinne ausfallen können, die wir mit intelligent gewählten Algorithmen erreichen können. Sie werden allerdings auch feststellen, dass es Fälle gibt, in denen wir tatsächlich einen signifikanten Aufwand haben, der sich nicht weiter reduzieren lässt.

Fazit:

Wenn wir gute Software entwickeln wollen, dann brauchen wir Konzepte und Methoden, die uns im Alltag überflüssig erscheinen würden, weil sie dort in aller Regel nur zusätzlichen Aufwand bedeuten. Der Bereich der **Praktischen Informatik** bietet mit „Algorithmen und Datenstrukturen" einen passenden Einstieg an.

2.4.1 Noch mehr Bezug zur Realität

Wenn Sie diesen Kurs erfolgreich abgeschlossen haben, dann werden Sie dementsprechend verstehen, warum es nicht genügt, eine Programmiersprache gut zu verstehen, um auch gute Software zu entwickeln.

Gleichzeitig ist dieser Kurs lediglich ein Einstieg in diesen Bereich. Hier setzen wir fast durchgehend voraus, dass die n Elemente, die unser Algorithmus bearbeiten soll sich während des Ablaufs nicht ändern. In der Praxis ist das aber leider nur selten der Fall. Deshalb kann dieser Kurs nur ein Einstieg in dieses wichtige Gebiet sein.

2.4.2 Sortieren mit Parallelprozessoren

Wenn Sie lesen, dass ein Computer vier Kerne hat und das jeder dieser Kerne mit 4 GHz arbeitet, dann könnten Sie auf die Idee kommen, dass der Computer effektiv mit 16 GHz (also viermal so schnell) arbeiten kann. Sie würden sich vielleicht noch denken, dass da ein wenig Verwaltungsaufwand abgezogen werden müsste. Doch selbst dann könnten Sie zumindest denken, dass solch ein Computer doch im Regelfall praktisch jedes Programm deutlich schneller ablaufen lassen müsste als ein Prozessor mit nur einem Kern.

Leider liegen Sie damit falsch: Wenn die ProgrammiererInnen eines Programms dieses Programm nicht explizit für Parallelprozessoren erweitert haben, dann läuft es selbst auf einem Computer mit 50.000 Kernen noch genauso langsam/schnell wie auf einem ansonsten baugleichen Computer mit nur einem Kern.

Nehmen wir einen Vergleich: Stellen Sie sich vor, Sie haben in Ihrer Garage zwanzig Autos, die baugleich sind. Können Sie deshalb zwanzig Mal

so schnell zum Supermarkt fahren, als wenn Sie nur einen davon hätten? Natürlich nicht. Aber wenn Sie 19 Freunde hätten, die alle genauso schnell wie Sie einkaufen können und Sie jedem von Ihnen einen Wagen geben würden, dann könnten Sie in der gleichen Zeit zwanzig mal so viele Einkäufe erledigen, als wenn Sie nur einen Wagen hätten.

Wie kommen wir jetzt von diesem Vergleich zu Programmen und parallelen Prozessoren? Ganz einfach: Ein Programm muss in unabhängige Teile zerlegbar sein, damit wir es auf mehreren Kernen eines Computers ablaufen lassen können. Und in unserem Beispiel gibt es eine Sache, die wir in unabhängige Teile zerlegen können: Die zu beschaffenden Einkäufe.

Aufgabe:

- Begründen Sie, warum Bubble Sort auf Parallelprozessoren schneller laufen kann als Insertion Sort.
- Rechnen Sie die folgende Aufgabe (a) für einen 10-Kern-Prozessor, (b) einen 50-Kern-Prozessor und (c) einen 100-Kernprozessor durch:

  Berechnen Sie die Dauer, die ein Bubble Sort Algorithmus auf dem genannten Prozessor (4 GHz-Takt) braucht, um seine Aufgabe (Sortieren von 100 Elementen) abzuschließen. Haben Sie eine realistische Chance, das er diese Aufgabe vor Ihrem Eintritt in die Rente abgeschlossen hat? (Schlagen Sie ggf. bei den Aufgaben zu Insertion Sort nach; dort stehen schon einige Teilberechnungen.)
- Warum macht es bei diesem Beispiel keinen Unterschied, ob der Prozessor 50 oder 100 Kerne hat?
- Warum macht es bei diesem Beispiel keinen Unterschied, ob der Prozessor einen 25-Kern-Prozessor oder einen 26-Kern-Prozessor hat?
- Warum macht es dagegen einen Unterschied, ob der Prozessor 24 oder 25 Kerne hat?

2.5 Programmierung von Bubble Sort

Wie schon beim Insertion Sort sind für jeden einzelnen Austausch zweier Elemente mehrere Operationen nötig. Wie gesagt vernachlässigen wir diese Details bei der O-Notation, weil dieser Mehrbedarf für alle Sortieralgorithmen gilt. Vergleichen Sie das Java-Fragment für Insertion Sort mit diesem, um zu verstehen, warum das so ist.

```
// Das zu sortierende Array ist: int[] values
```

```
void bubbleSort(){
do // while (elementSwitched);
{

for (int i = 0; i < values.length - 1; i+=2)
{
if (i == 0)
{
elementSwitched=false;
}
if (values[i] > values[i+1])
{
elementSwitched = true;
switchElements(i,j);
} // fi (values[i] > values[i+1])
} // rof (int i = 0; i < values.length - 1; i+=2)

for (int i = 1; i < values.length - 1; i+=2)
{
if (values[i] > values[i+1])
{
elementSwitched = true;
switchElements(i,j);
} // fi (values[i] > values[i+1])
} // rof (int i = 1; i < values.length - 1; i+=2)
} while (elementSwitched);
} // bubbleSort()

void switchElements(int i, int j)
{
int x = values[i];
values[i] = values[i+1];
values[i+1] = x;
}
```

Aufgabe:

- Nehmen Sie sich bitte ausreichend Zeit, um die Umsetzung von Insertion Sort und Bubble Sort in Java zu vergleichen.
- Berechnen Sie dazu als erstes die Laufzeitkomplexität der beiden anhand des Quellcodes.
- Erklären Sie ebenfalls schriftlich, wo die beiden Quellcodes sich unterscheiden und wo sie sich gleichen.

- Drücken Sie in eigenen Worten aus, was letztlich dazu führt, dass beide eine Laufzeitkomplexität von O(n^2) haben.
- Berechnen Sie S(n)
  (a) auf einem 5-Kern-Prozessor
  (b) auf einem 10-Kern-Prozessor,
  wenn der Algorithmus diese auch ausnutzen kann. Was schließen Sie aus Ihren Ergebnissen?

2.6 Zusammenfassung zu einfachen Algorithmen

Sie haben jetzt zwei Algorithmen kennen gelernt, die zwar unterschiedlich ablaufen, aber in der Praxis sind beide meist nicht sinnvoll einsetzbar.

Sie wissen jetzt, dass es ein Wertungskriterium gibt, um Algorithmen zu vergleichen, eben die O-Notation.

Sie sind im Stande, Algorithmen anhand der O-Notation zu vergleichen.

Sie wissen aber auch, dass Sie mithilfe der O-Notation nur einen Aussage über den relativen Zeitbedarf verschiedener Algorithmen treffen können.

Ihnen ist insbesondere bewusst, dass die O-Notation nichts darüber aussagt, wie groß der Speicherbedarf eines Algorithmus ausfällt.

Langfristig werden Sie erkennen, dass wir bei den beiden Komplexitäten (Laufzeit versus Speicherbedarf) ein Dilemma vorfinden, dass eine gewisse Analogie zur Heisenbergschen Unschärferelation aufweist: So wie wir uns im Bereich der Elementarteilchen entscheiden müssen, ob wir die Position oder die Geschwindigkeit eines Elementarteilchens mit (nahezu) beliebiger Präzision feststellen wollen, müssen wir uns bei Algorithmen und Datenstrukturen entscheiden, ob wir eine hohe Effizienz beim Zeitbedarf oder beim Speicherbedarf erreichen wollen. Beides zusammen geht nicht.

Kapitel 3

Laufzeit-effiziente Algorithmen

Das letzte Kapitel diente vorrangig dazu, Ihnen zu zeigen, dass der naheliegende Weg, um Daten zu verarbeiten oftmals ein desaströser Weg ist. Dennoch sind solchen einfachen Sortieralgorithmen durchaus ein sinnvolles Mittel, wenn Sie wissen, dass Sie sie in einem Programm nur für das Sortieren sehr kleiner Mengen (z.B. fünf Elemente) nutzen werden.

Schauen wir uns nun zwei Algorithmen an, die bezüglich der Laufzeit in den meisten Fällen wesentlich effizienter als Insertion Sort und Bubble Sort sind. Aber genau wie bei Insertion Sort und Bubble Sort gilt auch hier, dass diese Sortieralgorithmen für Arrays definiert sind.

Die beiden folgenden Algorithmen werden meist in der Gruppe der **Devide and Conquer Algorithmen** zusammen gefasst, da mit dieser Bezeichnung ein wesentlicher Charakter dieser Algorithmen zusammen gefasst wird.

3.1 Merge Sort

In einem Zwischenschritt geht Merge Sort genauso vor wie Bubble Sort. Ansonsten haben die beiden aber kaum etwas gemein: Nach einigen vorbereitenden Schritten werden auch bei Merge Sort Elemente paarweise verglichen und sortiert. Aber im Gegensatz zu Bubble Sort passiert das für jedes Element bei Merge Sort so nur ein einziges Mal.

Teil II

Verkettete Listen

Teil III

Bäume

Teil IV

Betriebssysteme

Stichwortverzeichnis

Algorithmen
    Devide and Conquer, 33
    Merge Sort, 33
Algorithmus
    NP-hart, 20
    Sortieralg., 15
Array, 13
    dynamisch typisiert, 14

Codierung, 12
Computerarchitektur, 7

Datenstruktur
    Array, 13

Kommunikationstechnik, 12
Komplexität
    Laufzeit, 18
    O-Notation, 18
Komplexitätsklassen, 18

Merge Sort, 33

NP-hart, 20

O-Notation, 18

S(n), 25
Sortieralgorithmen
    Insertion Sort, 16
Speicher, 8
    Addresse, 8

Technische Informatik, 7