Informatik II
-
Algorithmen und Datenstrukturen
-
Betriebssysteme

Markus Alpers
B.Sc. und Ausbilder f. Industriekaufleute

8. Juli 2016

Inhaltsverzeichnis

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, es 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 auch nochmal 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. Ein letzter typischer Fehler von reinen ProgrammiererInnen besteht darin, dass Sie häufig dazu neigen, sich nicht das Problem anzusehen, sondern zu versuchen, eine Lösung zu entwickeln, die sie mit ihrer Lieblingsprogrammiersprache hinbekommen. Warum das keine gute Idee ist haben Sie bereits in der Veranstaltung „Einführung ins Programmieren“ gelernt.

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 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 einer 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.

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.

Die vier Algorithmen, die wir in diesem Teil des Buches untersuchen setzen alle voraus, dass sie auf einem Array ausgeführt werden.

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.)

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) sortieren soll.

Wie Sie weiter oben gesehen haben, sprechen wir fürs erste nur über Arrays bzw. Algorithmen für Arrays und die Komplexität dieser Arrays.

Kapitel 2

Einfache Sortieralgorithmen

Aufgabe:

- Notieren Sie eine Methode, wie Sie eine solche Highscore-Liste sortieren könnten. Die Reihenfolge soll dabei der Punktstand der SpielerInnen sein.

  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 mit 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 sortieren Sie dann davor oder danach ein. Danach sind die ersten beiden Karten sortiert. Wir haben bis hierher also 0 + 1 Vergleichsoperationen.
- 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 die vierte und fünfte Karte 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. (Und 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
  - Die Variable n
  - log_2 n (in der Informatik meist als log n abgekürzt)
  - 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 Bereichen 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.

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

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:

- Berechnen Sie jetzt diese 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.)

  - **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 dauerts 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.

Diese Aufgaben sollten vorrangig dazu dienen, dass Sie ein Gefühl für die verschiedenen Klassen der O-Notation bekommen.

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, das in der Praxis aber relevant werden kann: Ausführung von Algorithmen auf Parallelprozessoren. Wir 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 mehrere Operationen nötig. Diese Details 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 jedem Fall vollkommen inakzeptabel. Und selbst eine Laufzeit von O(n) wäre schlecht. Doch bevor wir zu Algorithmen mit einer Laufzeit von kleiner als O(n) kommen, schauen wir uns noch einen **einfachen Sortieralgorithmus** an:

2.2.3 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 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,
  das dritte mit dem vierten,
  das fünfte mit dem sechsten usw.
  und vertauschen jeweils, wenn nötig.

- Dann vergleichen wir wieder paarweise, aber dieses Mal:

  Das zweite mit dem dritten,
  das vierte mit dem fünften,
  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 effizient eine Aufgabe zu lösen (z.B. die Verteilung von Aufgaben bei der Organisation einer Feier) 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.

Wenn wir also gute Software entwickeln wollen, dann brauchen wir neue Konzepte und Methoden. Und der Bereich der **Praktischen Informatik** bietet mit „Algorithmen und Datenstrukturen“ einen passenden Einstieg an.

2.3.1 Noch mehr Bezug zur Realität

Wenn Sie diesen Kurs erfolgreich abgeschlossen haben, dann werden Sie dementsprechend verstehen, warum es nicht genügt, eine Programmier-

sprache 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.

2.3.2 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 wenige 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 10-Kern-Prozessor, einen 11-Kern-Prozessor oder einen 12-Kern-Prozessor hat?

2.4 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.
- Formulieren Sie aus, was letztlich dazu führt, dass beide eine Laufzeitkomplexität von O(n^2) haben.

2.5 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 im Bereich der Elementarteilchen uns 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 Speicherbedarf 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

Betriebssysteme

Stichwortverzeichnis