Kerngebiet Algorithmen

Lukas Prokop

1. – 11. Juni 2009

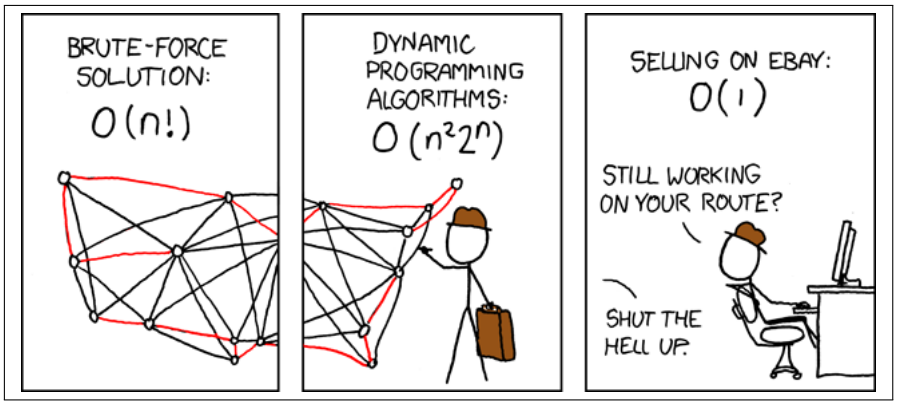
Dank an
Prof. Schmidhofer

"Beware of bugs in the above code;
I have only proved it correct, not tried it."

"A mathematical formula should never be *owned* by anybody!
Mathematics belong to God."

"Any inaccuracies in this index may be explained by the fact,
that it has been sorted with the help of a computer."
(all quotes above by Donald Ervin Knuth)

"A man provided with paper, pencil, and rubber, and subject to
strict discipline, is in effect a universal machine"
(Alan Turing)



"Travelling Salesman Problem" `http://xkcd.com/399/`

Inhaltsverzeichnis

Kapitel 1

Der Begriff Algorithmus

> "Ein Algorithmus ist die schrittweise Anleitung zur Lösung eines Problems. Idealerweise benötigt der Algorithmus wenig Ressourcen und hat eine endliche Laufzeit. Ein Programm ist die Realisierung eines Algorithmus in einer Programmiersprache."

Der Begriff Algorithmus stammt vom zentralasiatischen Mathematiker Muhammed al-Chwarizmi (etwa * 783 † 850) ab. Im Mittelalter wurde sein Ausdruck "Dixit Algorismi" ins Lateinische übersetzt: algorismus. Algorithmen sind das zentrale Element der theoretischen Informatik, die sich mit der Komplexitätstheorie und der Berechenbarkeitstheorie befasst.

1.1 Der erste Algorithmus

Charles Babbage (siehe Spezialgebiet Kryptologie) entwickelte den "Analytical Engine" (Rechenmaschine). Seine Mitarbeiterin Ada Lovelace ging als erste Programmierer(in) in die Geschichte ein, weil sie die erste Programmiersprache "ADA" formalisierte und einen Algorithmus zur Berechnung von Bernoulli-Zahlen entwickelte. Weil Babbage's Engine (wegen fehlender finanzieller Mittel) nicht fertig gestellt wurde, wurde der Algorithmus nie implementiert.

Der Begriff gewinnt ab dem 20. Jahrhundert (mit dem Aufkommen von Rechenanlagen) an Bedeutung. Bedeutende Wissenschaftler waren Alan Turing, Alonzo Church, Andrei Markow und Naom Chomsky. Sie lieferten wichtige linguistische Arbeiten, die zur Weiterentwicklung von Programmiersprachen und der mathematischen Problemlösung führten. Besonders Alan Turing möchte ich vorherheben, der die Komplexitätstheorie durch seine Turingmaschine begründete, zeigte dass alle Methoden (Lamda-Kalkül, Turingmaschine, Registermaschinen, Markow-Algorithmen) gleich mächtig sind und Wikipedia[1] zitiert aus seinen Arbeiten:

> Eine Berechnungsvorschrift zur Lösung eines Problems heißt genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt.
>
> Daraus sind folgende Eigenschaften ableitbar:
>
> 1. Das Verfahren muss in einem endlichen Text eindeutig beschreibbar sein (Finitheit)

[1] http://de.wikipedia.org/wiki/Algorithmus#Turingmaschinen_und_Algorithmusbegriff

2. Jeder Schritt des Verfahrens muss tatsächlich ausführbar sein (Ausführbarkeit)
3. Das Verfahren darf zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen (Dynamische Finitheit, Platzkomplexität)
4. Das Verfahren darf nur endlich viele Schritte benötigen (Terminierung, Zeitkomplexität)

Erweiterte Eigenschaften:

1. Der Algorithmus muss bei gleichen Voraussetzungen das selbe Ergebnis liefern (Determiniertheit)
2. Die nächste anzuwendende Regel im Verfahren ist zu jedem Zeitpunkt eindeutig definiert (Determinismus)

1.2 Ein Beispiel

Ein bekanntes Beispiel bildet die sogenannte "Identische Abbildung" in der Mathematik. Eine Funktion gibt ihren Eingabeparameter wieder zurück. Eine Funktion f ist also definiert durch...

$$f(x) = x$$

Wenn wir beispielweise die ganze Zahl 5 als Argument übergeben, erhalten wir $f(5) = 5$ zurück. Der entsprechende Algorithmus könnte dann beispielweise wie folgt aussehen:

1. Empfange den Parameter a
2. Initialisiere die Variable b als 0 ($b = 0$)
3. Solange a ungleich 0
   inkrementiere b und dekrementiere a
4. Gib b zurück

Übergeben wir der Funktion (bzw. dem Algorithmus) den Parameter 3, wird er wie folgt ausgeführt:

1. $a = 3$
2. $b = 0$
3. $a \neq 0$ (wahr)
   $b = 0 + 1 = 1$
   $a = 3 - 1 = 2$
4. $a \neq 0$ (wahr)
   $b = 1 + 1 = 2$
   $a = 2 - 1 = 1$
5. $a \neq 0$ (wahr)
   $b = 2 + 1 = 3$
   $a = 1 - 1 = 0$
6. $a \neq 0$ (falsch)
   return 3

1.3 Ein bekannteres Beispiel

Ein bekannteres Beispiel ist die Division, wie man sie in der Grundschule lernt.

```
86      : 3 = 28.666
26
 20
  20
   20
    2 Rest
```

Der Algorithmus lautet formuliert:

1. Nehme die zwei Zahlen a, b entgegen (b als Divisor)
2. Wenn b gleich 0, dann Exception/Error werfen ("Divsion by zero")
3. Solange (Stellen noch übrig) oder (Rest periodisches Verhalten aufweist)
   nehme die größte Stelle (hier: Zehnerstelle) und dividiere sie durch b
   notiere das Ergebnis nach dem Gleichzeichen
   Schreibe den Rest der letzten Division unter diejenige Zahl
   Wenn übrige Stelle vorhanden
   (a) dann schreibe sie hinunter, sodass sie Einerstelle des neuen Wertes wird

   Wenn keine übrige Stelle vorhanden
   (a) dann schreibe eine Null hinunter
   (b) wenn das Ergebnis noch kein Kommazeichen enthält
      i. notiere ein Kommazeichen im Ergebnis
4. definiere die nächste Stelle als die "größte Stelle"
5. Gib das Ergebnis zurück

1.4 Darstellungen

Oft ist es hilfreich einen Algorithmus visuell darzustellen. Dabei definiert man bereits Elemente, die klar machen, welche Werkzeuge wir zur Formulierung von Algorithmen benötigen. Man kann verschiedene Darstellungen unterscheiden.

- Flussdiagramm
- Struktogramm (Nassi-Shneiderman-Diagramm)
- Aktivitätsdiagramm

- Jackson-Diagramme

Es gibt verschiedene ISO und DIN-Spezifikationen, die diese Abläufe festlegen. Die UML (Unified Modeling Language) Spezifikation ist seit den 90ern vertreten und bietet Darstellungsmöglichkeiten für alle Einsatzgebiete. Die UML umfasst die oben angesprochenen Darstellungen. Im folgenden möchte ich zwei Diagramme besprechen.

Das Struktogramm (bzw. Nassi-Shneiderman-Diagramm nach den beiden Softwareentwicklern Isaac Nassi und Ben Shneidermani benannt) teilt Algorithmen in rechteckige Blöcke auf. Der ganze Algorithmus wird in ein großes Rechteck eingebettet. Die einzelne Schritte werden nur als kleinere Rechtecke eingetragen. Für bedingte Anweisungen wird ein Rechteck definiert, das in drei Dreiecke zerlegt wird. In das mittlere Dreieck wird die Entscheidungsfrage formuliert und den beiden anderen die mögliche Antwort auf die Frage. Die beiden Antwortsdreiecke schließen mit ihren jeweiligen weiteren Schritten an.

Für Schleifen wird der beeinflusste Quelltext eingerückt. Ein Aufruf eines Unterprogramms wird durch ein Rechteck mit doppelten Rändern symbolisiert.

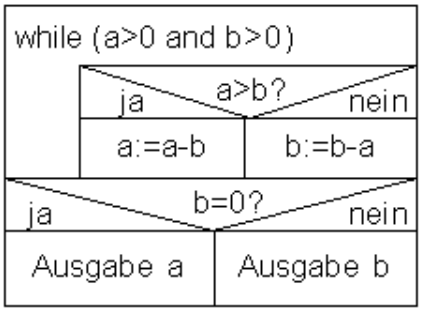


Abbildung 1.1: Euklidischer Algorithmus im NassiShneiderman-Diagramm

Dank an Wikipedia für dieses Bild.

Und mein persönliche Liebling ist das Flussdiagramm (auch Programmablaufsplan genannt). Hierbei werden Aktionen durch geometrische Formen dargestellt und durch Pfeile verbunden.

Ein ovaler Körper steht für Grenzpunkte (Start, Stop). Ein Pfeil oder eine Linie stellen eine Relation her (Übergabe von Parametern oder Markierung als nachfolgender Schritt). Ein Rechteck steht für eine Operation (zB einer Zuweisung). Ein Rechteck mit doppelten vertikalen Rändern für einen Aufruf eines Unterprogramms, eine Raute für eine bedingte Anweisung (Verzweigung) und ein Parallelogramm für Ein- und Ausgabe.

Dank an Wikipedia für dieses Bild.

Was wir an dieser Stelle festhalten wollen: Die Algorithmen werden *schrittweise* formuliert. Dieses Wort findet sich auch in der Definition von Algorithmen am Anfang des Dokuments wieder. Die Idee der schrittweisen Anleitung entspricht den Idealen der imperativen Programmierung, wo Befehle Schritt für Schritt abgearbeitet werden. Die imperative Programmierung ist ein sehr frühes Programmierparadigma und ist heute Element jeder Programmiersprache. Daraus folgt, dass jeder (klassische) Algorithmus in jeder Programmiersprache (bzw. Programmiersprachen jeder Generation) implementiert werden kann. Ein größeres Problem bei der Implementierung stellen Ressourcenverbrauch und Laufzeit dar. Die Formulierung eines Algorithmus in einer Programmiersprache selbst nahezu nie. Es ist schließlich auch Aufgabe einer Programmiersprache

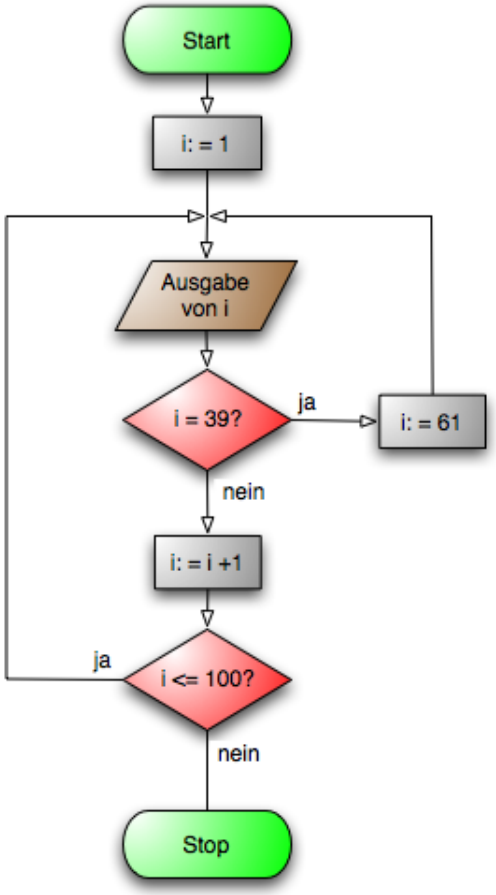


Abbildung 1.2: Beispiel für ein Flussdiagramm

die Arbeit des Programmierers zu erleichtern.

1.5 Das Urheberrecht

Gemäß dem Patentrecht lassen sich Algorithmen patentieren. Viele Probleme werfen mehr als eine Lösung auf und Wissenschaftler stecken viel Arbeit in die Entwicklung von Algorithmen und wollen deshalb ihr Werk patentieren lassen. So ist dies zum Beispiel beim MP3-Komprimierungsverfahren der Fall, wo Elemente des Verfahrens geschützt sind. Implementierungen des MP3-Algorithmus sind also nicht gestattet und die freie Verbreitung von MP3-Programmen ist nicht möglich.

Kapitel 2

Aus der Komplexitäts- und Berechenbarkeitstheorie

Die Komplexitätstheorie versucht die Komplexität von Algorithmen zu analysieren bzw. Algorithmen zu klassifizieren. Die Berechenbarkeitstheorie versucht hingegen nur die Frage zu beantworten welche Probleme "lösbar" sind.

2.1 Klassische Probleme der Informatik

2.1.1 Problem des Handlungsreisenden

Das Problem findet sich in nahezu allen Bereichen. Im Design von Mikrochips, in der Logistik und in der Entwicklung Routenplanern.

Es sind mehrere Zielorte angegeben. Zum Beispiel muss der Handlungsreisende zu allen Standorten von Firmen anreisen, um sich mit ihnen zu treffen. Dazu möchte er möglichst wenig Zeit aufwenden. Es ist also sein Ziel die Reihenfolge der Besuche so zu wählen, dass die Gesamtreisezeit minimal gehalten wird und die Rückkehr an den Ausgansort möglichst schnell erfolgt. Das Problem: Er darf nicht alle Wege zuvor abfahren und ihre Länge messen.

Dieses Problem ist unlösbar.

2.1.2 Speisende Philosophen

Das Problem findet sich bei der Prozessverwaltung bzw. Verwaltung von Prozessen.

Die 5 fünf Philosophen sitzen am Tisch und philosophieren. Wird einer der Philosophen hungrig wird, ergreift er die Gabel links und rechts und beginnt zu essen. Leider stehen nur 5 Gabeln für 5 Philosophen zur Verfügung, aber solange nur einzelne Personen essen möchte, stellt dies kein Problem dar. Ein Problem wird es, wenn sich alle zugleich entschließen zu essen. Dann greifen alle Philosophen zugleich zu der Gabel links und können die Gabel rechts nicht beanspruchen (weil der rechte Nachbar sie in der linken Hand hat). Als Folge halten alle ihre Gabel und warten auf eine zweite. Es kommt zu einem unveränderlichen Zustand und die Philosophen verhungern.

Das Warten auf Gabeln ist analog zum Warten auf Ressourcen von anderen Prozessen. Man

spricht von einem "dead lock"

2.1.3 Rucksackproblem

Das Problem findet sich in der Logistik und allgemein in Bereichen mit beschränkten Kapazitätsverhältnissen.

Man haben einen Rucksack, der insgesamt 100kg trägt. Wir haben jetzt die Gewichte 75kg, 65kg, 35kg, 12kg, 14kg, 4kg, 5kg, 2kg, 19kg und 20kg. Finde die Permutation, wo das Gewicht des Rucksacks möglichst groß wird, jedoch nicht die Schranke 100kg überschreitet.

In der Kryptologie ist dieses Problem unter dem Begriff Subset-Sum bekannt. Dieses Problem ist am besten durch Backtracking lösbar.

2.1.4 Damenproblem

Es sollen acht Damen so auf einem Schattbrett verteilt werden, dass keine Dame eine andere "bedroht". Gemäß den Schachregeln bedroht eine Dame eine Figur, wenn die Figur in einer geraden oder diagonalen Linie zur Dame steht.

Das Problem wird meist durch rekursives Backtracking gelöst. Die Anzahl der Lösungen wächst etwas schneller als exponentiell mit der Brettgröße an. Auf einem 6×6-Brett gibt es interessanterweise weniger Lösungen als auf einem 5×5-Brett.

2.1.5 Landkartenfärbungsproblem

Du hast eine Landkarte vor dir und musst die Länder mit verschiedenen Farben eindeutig einfärben. Zwei benachbarte Länder dürfen also nicht die selbe Farbe tragen. Reduziere die Anzahl der verwendeten Farben auf ein Minimum.

Unter den Bedingungen, dass zwei F̈lachen mit einem gemeinsamen Punkt nicht als benachbart gelten und ein Land keine Exklaven besitzt, ist die Frage mit 4 zu beantworten. Das Problem ist heute als Vier-Farben-Problem bekannt.

Das Besondere ist, dass Mathematiker im 19/20. Jahrhundert verschiedene Beweise für das Problem formulierten, die sich gegenseitig wiedersprachen. Mit dem Einsatz von Computer wurde zum ersten Mal gezeigt, dass der Computer immer bloß vier Farben brauchte. Die Mathematiker erkannten diese Tatsache nicht an, da das Problem nicht von Menschenhand gelöst wurde.

2.1.6 Beziehung von Problemen und Algorithmen

Diese Probleme sind entweder selbst algorithmisch zu lösen, oder sie eignen sich als verallgemeinerter Problemaufriss für reale Beispiele.

Probleme (dieser Art) allgemein lassen sich mathematisch formulieren und beweisen. Algorithmen sind eine Lösungsanleitung um diese Probleme in jeder Situation effizient und effektiv zu lösen.

2.2 Turingmaschine

Die Church-Turing-These (nach Alonzo Church und Alan Turing) behauptet, dass alle mathematischen Probleme (die ein Mensch lösen kann) auch von einer Turingmaschine gelöst werden können. Die Annahme eine Turingmaschine könnte alle mathematischen Probleme lösen ist allerdings bewiesen falsch. Alan Turing bewies, dass das Halteproblem nicht mit der Turingmaschine entscheidbar ist.

Das Halteproblem ist Algorithmus, der nicht terminiert werden kann. Entweder der Algorithmus selbst terminiert. In dem Fall springt der Algorithmus in eine while-Schleife, die nicht terminiert und keine Befehle ausführt. Geben wir an, dass der Algorithmus nicht terminiert, so terminiert der Algorithmus sofort. Dies endet in einem Widerspruch. Alan M. Turing konnte 1936 beweisen, dass es keine Turingmaschine gibt, die das Problem für alle Eingaben löst, wobei das Problem mathematisch als korrekt anzusehen ist. Es ist das erste bewiesene Entscheidungsproblem, welches nicht algorithmisch bewiesen werden kann.

```
def halting_test(program, parameter):
    if program_halts(programm(parameter)):
        return True
    else:
        return False

def test(program):
    while program_halts(halting_test(program, program)):
        pass # do nothing

test(test)
```

Bei der Turingmaschine handelt es sich um eine reduzierte Rechenmaschine, die nur 3 Operationen ausführen kann: Schreiben, Lesen und Bewegen. Mit diesen Operationen lassen sich jedoch Addition, Subtraktion und damit der Großteil der mathematischen Funktionen simulieren. Man könnte es sich als endloses Band vorstellen, welches sich mit Einsen und Nullen beschreiben lässt. Ein Lesekopf führt jetzt entsprechend einem Algorithmus die Befehle Schreiben, Lesen und Bewegen aus.

Lassen sich Probleme auf einer Turingmaschine lösen, kann jeder Computer das selbe Problem auch lösen. Es dient als Gedankenexperiment über die Fähigkeitsgrenze von logisch-algorithmischen behandelten Problemen.

2.3 Klassifikationskriterien für Algorithmen

Die Turingmaschine eignet sich ideal in der Berechenbarkeitstheorie, um Entscheidungen über die Berechenbarkeit von Problemen zu treffen. Alternative Maschinenmodelle sind beispielweise die Registermaschine, der endliche Automat und der Kellerautomat.

Die folgenden Begriffe werden sowohl von der Berechenbarkeitstheorie wie auch Komplexitätstheorie verwendet. Teilweise wurden sie oben angesprochen:

Finitheit allgemein: Endlichkeit

- **statisch** die Beschreibung des Algorithmus ist endlich

dynamisch es werden endlich viele Ressourcen gebraucht

Terminiertheit ein Algorithmus terminiert; ein Problem wird in endlicher Zeit gelöst. Das Halteproblem zeigt, dass es keinen Algorithmus gibt, der sagen kann, ob ein Programm für die Eingabe terminiert. Das gleichmäßige Halteproblem besagt, dass ein Algorithmus selbst nicht beweisen kann, ob er überall (unter beliebigen Umständen) terminiert

Determinismus Es treten nur definierte (reproduzierbare) Zustände auf. Auf eine Anweisung im Algorithmus folgt unter den gleichen Voraussetzung immer eine genau definierte Anweisung. Ein deterministischer Algorithmus ist immer determiniert, aber es gibt nichtdeterministische Algorithmen, die trotzdem determiniert sind.

Determiniertheit Ein Algorithmus ist deterministisch, wenn er unter allen Voraussetzungen das selbe Ergebnis liefert.

Nichtdeterminiert Ggt. von Determinismus. Nichtdeterministische Algorithmen sind randomisiert (zB AKS-Primzahltest)

Las-Vegas randomisierte Algorithmen, die nie ein falsches Ergebnis liefern

Monte-Carlo randomisierte Algorithmen, die eventuell ein falsches Ergebnis liefern (zB Miller-Rabin-Primzahltest)

Ein Kriterium möchte ich hinzufügen, welches vorwiegend nur bei Sortieralgorithmen wichtig ist: Stabilität. Als stabil wird ein Algorithmus bezeichnet, der ein Ergebnis ausgibt, das sich an der Eingabe orientiert. In einer Liste mit den Zahlen [1, 4, 1, 3] könnte der erste Einser an die Position 1 und der zweite Einser (aktuell an der Position 2) an die Stelle 0 rücken (instabil). Bleibt die Reihenfolge jedoch wie in der Urliste erhalten (0→0, 2→1), spricht man von Stabilität.

2.4 Erfassen der Komplexität von Algorithmen

Um die Schwere und den Aufwand von Algorithmen zu analysieren, müssen wir die Landau-Notation kennen lernen.

Notation	Wirkung
$f \in \mathcal{O}(g)$	f wächst nicht wesentlich schneller als g
$f \in \mathrm{o}(g)$	f wächst langsamer als g
$f \in \Omega(g)$	f wächst mindestens so schnell wie g
$f \in \omega(g)$	f wächst schneller als g
$f \in \Theta(g)$	f wächst genauso schnell wie g

Diese Definition ziehen wir heran, um die Laufzeit eines Algorithmus' zu beschreiben. x ist jeweils das Argument der Funktion f. Alle "Bedeutungen" sind "ungefähr".

Bezüglich sehr großer Zahlen ist diese Liste geordnet ($x!$ wächst bei großen Zahlen ganz schnell). Wichtig ist jedoch, dass diese Landau-Symbole keine Zahlen ausgeben, die sich in die (ideale) Laufzeit umrechnen lassen, sondern bloß um Richtwerte, die ein Wachstum definieren. So ist beispielweise die Basis des Logarithmus komplett egal. Die meisten Sortieralgorithmen haben einen Speicherverbrauch von $\mathcal{O}(1)$, da sie bei einer Eingabe von n Elementen n Plätze benötigen. Bei einer Eingabe von $n+1$ Elementen benötigen sie $n+1$ Plätze. Das Wachstum ist konstant und wird deshalb mit 1 beschrieben.

Notation	Bedeutung
$f \in \mathcal{O}(1)$	f besitzt eine Schranke (die unabhängig vom Argument ist)
$f \in \mathcal{O}(\log x)$	doppeltes Argument, f wächst um konstanten Wert
$f \in \mathcal{O}(\sqrt{x})$	f doppeltes f, vierfaches x
$f \in \mathcal{O}(x)$	lineares Wachstum, doppeltes x, doppeltes f
$f \in \mathcal{O}(x \log x)$	super-lineares Wachstum
$f \in \mathcal{O}(x^2)$	f vierfach, wenn x doppelt
$f \in \mathcal{O}(2^x)$	$x + 1$ resultiert in f doppelt
$f \in \mathcal{O}(x!)$	f wächst um das x-fache, wenn $x - 1$ zu x erhöht

Beispiele sind beispielweise das binäre Suchen (logarithmisch), lineare Suchen (linear), Quicksort (leicht überlineare Komplexität), Sortieren durch Auswahl oder Bubblesort (quadratisch) oder Problem des Handlungsreisenden oder Türme von Hanoi (exponentiell)

2.4.1 worst and best case

Neben dem average-Case (wie Algorithmen sich im Durchschnitt verhalten), wird auch der best und worst case in Landau-Notation festgehalten. So gibt es Sortieralgorithmen, die zwar eine total ungeordnete Liste (worst case) in sehr geringer Zeit ordnen können, doch beginnen eine geordnete Liste (best case) ebenfalls durcheinander zu würfeln. Der average-case reicht nicht aus, um die Komplexität eines Algorithmus' zu erfassen. Einige Algorithmen haben jedoch bei best, worst und average case das selbe Verhalten (bzw. Komplexität)

2.4.2 Komplexitätsklasse

Eine Komplexitätsklasse umfasst Probleme, die alle die selbe Komplexität aufweisen. Am bekanntesten ist die Klasse NP (nichtdeterministisch polynomielle Zeit). Nichtdeterminismus bezeichnet (wie oben erwähnt) Algorithmen, dessen nächster Schritt nicht eindeutig festgelegt ist. Polynomiell bezieht sich auf die Eingabemenge und bezeichnet Algorithmen, die mit steigendem Eingabeparameter eine steigende Laufzeit besitzen, die unterhalb der von Polynomen liegt. Ein Problem ist genau dann NP-vollständig, wenn dessen Eigenschaften innerhalb der NP-Komplexitätsklasse liegen.

Das Rucksackproblem und das Problem des Handlungsreisenden sind NP-vollständig.

Eine andere Komplexitätsklasse ist P (aka PTIME). Sie umfasst Probleme, die polynomielle Eigenschaften aufweisen, aber deterministisch sind.

2.5 Lösungsansätze

2.5.1 Heuristik

Unter Heuristik versteht man Algorithmen, die mit geringem Rechenaufwand und kurzer Laufzeit zulässige Lösungen für ein bestimmtes Problem liefern. Bei Algorithmen besitzt man den Anspruch auf optimale Lösungen in optimaler Laufzeit. Heuristische Algorithmen verzichten auf eine der Kriterien, um andere Eigenschaften zu gewährleisten.

Anwendung: Man entwickelt einen Algorithmus, um ein Problem zu lösen.

- Der Algorithmus hat eine sehr große Laufzeit
- Der Algorithmus liefert mehrere Lösungen
- Der Algorithmus liefert zuverlässig eine Lösung

Man schreibt jetzt den Algorithmus heuristisch, dann besitzt er beispielweise die folgenden Eigenschaften:

- Der Algorithmus hat eine vergleichsweise geringe Laufzeit
- Der Algorithmus liefert nur eine Lösung
- Der Algorithmus liefert eine zuverlässige Lösung

Als triviales Beispiel kann das Erfüllen einer mathematischen Gleichung herangezogen werden. Man verzichtet darauf alle Ergebnisse zu errechnen und erreicht dafür eine geringere Laufzeit.

2.5.2 Backtracking

Backtracking wurde bereits oben erwähnt. Einige der oben angesprochene Probleme sind durch Backtracking lösbar. Backtracking bezeichnet dabei den Lösungsansatz alle Lösungswege durchzuwandern und am Ende den besten Weg herauszusuchen (Minimum bzw. Maximum). zB Einen Algorithmus für das Damenproblem beginnt man damit eine Dame am Anfang an einen beliebigen Ort zu setzen. Danach setzt man die zweite Dame in der zweiten Reihe. Tritt ein Fehler (Dame bedroht Dame) auf, setzt man die zweite Dame zurück (backtrackt sie). Schrittweise versucht man so zu einer Lösung zu kommen. Bei einem Fehler geht man einen Schritt zurück. Kommt man zu einer validen Lösung, dokumentiert man das Ergebnis (beim Damenproblem kann man sogar Schachnotation verwenden). Am Ende vergleicht man die Ergebnisse und retourniert die optimale Lösung.

Das Problem des Handlungsreisenden wäre auch mit Backtracking lösbar, jedoch ist es laut Problemstellung nicht erlaubt jeden Weg abzufahren. Durch Backtracking steigt die Laufzeit erfahrungsgemäß enorm.

2.5.3 Teile und herrsche

Das Prinzip "divide and conquer" schreibt eine Lösung vor, wo das Problem in kleinere Teilprobleme aufgeteilt wird. Beispielweise arbeitet der Sortieralgorithmus Mergesort so, dass das Problem so weit geteilt wird, bis nur mehr Buchstabenpaare übrig bleiben. Diese Buchstabenpaare werden alphabetisch sortiert und dann mit den anderen Paaren zusammengeführt.

Kapitel 3

Bekannte Algorithmen

3.1 Ein Algorithmus im Detail: Mergesort

Mergesort ist ein Sortieralgorithmus nach dem Prinzip Teile und herrsche. Er teilt die – zu verarbeitende Liste – in mehrere Teile und sortiert die einzelnen Buchstaben, die in Teilen gespeichert sind.

```
def mergesort_rec(liste):
    if len(liste) <= 1:
        return liste
    else:
        mid = (len(liste) / 2)
        left, right = liste[0:mid], liste[mid:]
        return merge_it(mergesort_rec(left), mergesort_rec(right))

def merge_it(left, right):
    nlist = []
    while len(left) > 0 and len(right) > 0:
        if left[0] <= right[0]:
            nlist.append(left[0])
            del left[0]
        else:
            nlist.append(right[0])
            del right[0]
    while len(left) > 0:
        nlist.append(left[0])
        del left[0]
    while len(right) > 0:
        nlist.append(right[0])
        del right[0]
    return nlist
```

Mergesort ist ein ideales Beispiel, welches zeigt, dass Rekursion oftmals besser zum Programmieren geeignet ist, da der Algorithmus dadurch übersichtlicher und kürzer ist.

Der Algorithmus ist stabil und besitzt in jedem (worst, best, average case) Fall eine Komplexität von $\mathcal{O}(n \cdot \log(n))$. Durch sein Divide-and-conquer-Verhalten steigt der Wachstum des Speicherbedarfs bei einer wachsenden Eingabe von Elementen. Die Raumkomplexität wird mit $\mathcal{O}(n)$ beschrieben.

3.2 Euklidischer Algorithmus

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    int a, b, base = 10;
    char *endptr;
    int unsigned tmp;
    a = strtol(argv[1], &endptr, base);
    b = strtol(argv[2], &endptr, base);

    // euklidean algorithm
    while (b != 0) {
        tmp = a % b;
        a = b;
        b = tmp;
    }

    printf("greatest common divisor: %d \n", a);
    return 0;
}
```

Der euklidische Algorithmus errechnet den größten gemeinsamen Teiler zweier ganzer Zahlen. Man kann vier Varianten unterscheiden:

- klassisch, iterativ
- klassisch, rekursiv
- modern, iterativ
- modern, rekursiv

Der oben (in C) notierte Algorithmus ist modern und iterativ. Der klassische Algorithmus basiert auf der Idee, dass durch ständiges Subtrahieren des kleineren Wertes vom Gröseren am Ende die beiden Werte zusammenfallen, einer der Wert null wird und der andere Wert (wird zurückgegeben und) ist der der ggT (engl. gcD). Der moderne Algorithmus verwendet hier einfach den Rest der Division.

Bei Subtraktion handelt es sich um das Gegenteil der Addition. Bei einem Teiler handelt es sich um das Gegenteil eines Vielfachen. Addiert man einen Wert definiert oft, entsteht ein Vielfaches des Wertes. Diesen Weg zurückgedacht, ergibt den euklidischen Algorithmus

Eine Verbesserung des Euklidischen Algorithmus ist der Steinsche Algorithmus. Also muss er richtig implementiert werden, damit er schneller als der moderne iterative Euklid ist (wie ich unweigerlich erfahren musste).

3.3 Binäres Suchen

Interessant ist eine Erfahrung, die wohl die meisten Menschen unbewusst im Leben gemacht haben: Suchen. Suchen wir ein Wort in einem Wörterbuch, so werden wir nicht damit beginnen jedes Wort von Anfang an durchzugehen, sondern werden das Buch in der Mitte aufschlagen. An der Buchstabenangabe sehen wir, ob sich das Wort in der linken oder rechten Hälfte befindet. Hat man die entsprechende Hälfte gefunden, wird diese wieder in die Hälfte geteilt... und so weiter bis nur mehr ein Element übrig bleibt, welches mit dem Suchelement überein stimmt.

Die Idee jedes Wort der Reihe durchzugehen, bezeichnet man als lineares Suchen. Die Idee des Halbierens bezeichnet man als binäres Suchen. Binäres Suchen setzt eine geordnete Liste von Elementen voraus. Lineares Suchen nicht. Den best case für lineares Suchen bildet der Fall, dass sich das gesuchte Element am Anfang befindet. Den best case für binäres Suchen bildet der Fall, dass sich das gesuchte Element in der Hälfte befindet (das Element eines daneben bildet den worst case).

3.4 Weitere Algorithmen

Es gibt eine Vielzahl von Algorithmen. Im Folgenden folgt nur ein Auszug von Algorithmen, die mir schon vor der Beschäftigung mit dem Thema bekannt war. Ich gehe also davon aus, dass jene Algorithmen den meisten Informatikern bzw. Mathematikern bekannt sind. $\frac{1}{3}$ der Rechenzeit eines modernen Heimrechners beanspruchen Sortierungen. Deshalb ist dieses Feld besonders beliebt als Einstieg zu Erklärungen.

Man kann sich selbst einmal damit befassen, wie Algorithmen zu verfassen sind. Man hat eine ungeordnete Liste von Algorithmen vor sich. Jetzt möchte man diese Liste ordnen. Das menschliche Auge versucht hier einfach die größte Karte zu erkennen. Diese Karte wird entnommen und als letztes Element der neuen Liste (out-of-place) definiert. Aber dabei handelt es sich gar nicht um ein optimales Verfahren. Der Zeiger muss jedes Mal die gesamte Liste durchgehen und kann dann erst entscheiden, welches das x-größte Element ist. Dieser Algorithmus ist verkehrt (das kleinste Element wird ausgesucht) als Insertion-Sort bekannt.

Sortieralgorithmen (Sortieren der Elemente einer Liste) (znlE = zwei nebeneinander liegende Elemente):

Gnomesort ordne znlE. wenn geändert, Schritt zurück, sonst vor

Mergesort divide and conquer, teile alles in Buchstabenpaare auf und sortiere sie einzeln. Führe alles wieder zusammen

Quicksort ähnlich MergeSort

Selectionsort, Heapsort wähle das kleinste Element und stelle es an Position 0

Shellsort, Insertsort vergleiche znlE und verschiebe evtl. Element ganz nach vorne

Bubblesort vergleiche znlE und tausche sie, falls verkehrt. Wenn in einem Durchlauf etwas verändert wurde, dann fange wieder von vorne an, sonst bereits sortiert

Simplesort vergleiche znlE und sortiere bis alle korrekt liegen

Slowsort sortiere alles außer dem größten Element

Algorithmen aus der Informatik und Mathematik:

- String-Matching-Algorithmen (suchen Stichwörter in einer Sammlung von Texten)
- Gaussches Eliminationsverfahren (Lösen von Gleichung mit mehreren Bedingungen)
- Sieb des Eratosthenes (Erzeugung von Primzahlen)
- Sieb des Atkin (Optimierung des Sieb des Eratosthenes)
- Miller-Rabin-Test (Primzahltest)
- Erweiterter euklidischer Algorithmus ($ggT(a, b) = s \cdot a + t \cdot b$)

Kapitel 4

Implementierung von Algorithmen

4.1 Der Begriff Syntax

Jede Sprache besteht im Grunde aus zwei Elementen: dem Vokabular und der Grammatik. Eine Programmiersprache ebenfalls. Ein Vokabular einer Programmiersprache bezeichne ich in den folgenden Sektionen als "Schlüsselwort" (engl. keyword). Und die Grammatik einer Programmiersprache wird als Syntax bezeichnet. Es gibt Linguisten, die Programmiersprachen genau untersucht haben und teilweise den Begriff Syntax abgrenzt, aber für Nichtprogrammierer ist die Erklärung hinreichend eindeutig. Programmierer wissen sowieso, was man unter Syntax versteht.

4.2 Hello World

"Hello World" ist eine berühmte Phrase, die Programmiereinsteigern ans Herz gelegt wird. Hello World setzt voraus, dass man den Interpreter oder Compiler starten kann, was die Basis jedes Programmierens ist. Danach muss man noch die Struktur eines allgemeinen Programms beherrschen (zB Einbinden der notwendigen Bibliotheken zur stdout-Ausgabe) und es folgt ein Programm, welches den schlichten Text "Hello World" ausgibt. Es ist das erste Programm, welches man beim Erlernen einer neuen Sprache schreiben sollte. Da es bei jeder Programmiersprache um das Ausgeben von Daten geht, lässt sich Hello World in jeder gewöhnlichen (nicht unbedingt turing-vollständigen) Programmiersprache implementieren.

4.3 Werkzeuge aus der Programmierung

Wikipedia gibt nette Beispiele. Diese folgenden Beispiele sollen nur zeigen, welche Operationen abarbeiten können müssen.

einfache Operation "2·2"

sequentielle Notation "Bier auf Wein, lass es sein" (Reihenfolge wichtig!)

nebenläufig "C++ und python" (Reihenfolge egal; evtl. gleichzeitig)

parallel "mit Gabel und Messer essen" (kann nur gleichzeitig erfolgen)

nichtdeterministisch, nichtdeterminiert "Vanille- oder Schokoeis" (Ergebnis von Wahl abhängig)

Weitere Anweisungen, die Computer können müssen haben wir bei den visuellen Darstellungen von Algorithmen gesehen. Als Basis gehört dazu:

- Bedingte Anweisungen (wenn *das*, dann *das* sonst *das*)
- For-Schleife (Wiederholen einer Befehlsfolge mithilfe einer Laufvariable)
- While-Schleife (Wiederholen einer Befehlsfolge mit unbekannte Laufzeit)

4.3.1 Bedingte Anweisungen

Mithilfe der Logik wird ein Entscheidung getroffen, welche einen Wahrheitswerte oder sein Gegenteil liefert. Trifft die Entscheidung zu, so wird der nachfolgend definierte Codeblock ausgeführt (if). Es existieren auch immer Deklarationen für Blöcke die ausgeführt werden, wenn der Wahrheitswert nicht gegeben ist (else).

Bei Programmiersprachen findet man die bedingte Anweisung typischerweise unter dem Schlüsselwort *if*.

Eines meines der ersten Probleme beim Programmieren war es nicht zu verstehen, was sich ändern sollte. 3 ist immer kleiner als 5 ($3 < 5 \rightarrow$ true) und mein Vorname wird immer Lukas sein. Ich habe 3 Elemente definiert, die sich ändern können und auf das die meisten Programme reagieren müssen:

Umwelt, System der Computer (installierte Software, Betriebsystem, ...) und die Peripherie des Computers (Wärme der CPU, Festplatte...)

Umgebungsvariablen Zeit, Daten aus einer Datenbank

Benutzereingabe Passwort, Username, Auswahl von Elementen, Maus, Tastatur

4.3.2 Schleifen

Codeblöcken müssen manchmal mehrfach ausgeführt werden. Dafür verwendet man in der Programmierung Schleifen. Man unterscheidet zwei Arten:

for Es wird eine genau definierte Anzahl an Wiederholung vollführt. Zugleich läuft meist eine Laufvariable mit, die die aktuelle Zahl der Wiederholung zurückgibt (1, 2, 3, 4, 5...). Die Laufvariable wirst standardmäßig inkrementiert, könnte jedoch auch dekrementiert werden.

while Ein Codeblock wird ausgeführt, wobei noch niemand weiß wie oft dies erfolgen wird. Meist wird in dem Codeblock selbst entschieden, ob der Code nun aus der Schleife springen soll, oder nicht. Es gibt auch Schleifen, die niemals terminieren. Sogenannte Endlosschleifen finden bei Servern und der Computer warten mittels eine Endlosschleife ständig auf neue Eingaben von Maus und Tastaur.

Ebenfalls ist es wichtig auf Datentypen reagieren zu können. Jede Programmiersprache besitzt ein Containerelement, wo mehrere Elemente in einer Variable zusammengefasst werden. Es ist jetzt notwendig jedes Element dieses Container durchschreiten zu können. Hierfür wird oft das

Schlüsselwort *for* selbst angewandt (die weitere Syntax unterscheidet dann eine Schleife mit Zählvariable von einem Containerdurchlauf) oder *foreach*, wobei die Programmiersprachen natürlich die Schlüsselwörter selbst definieren dürfen (siehe Programmiersprachen wie Brainf*uck, Whitespace oder beatnik).

4.3.3 Logik

Die Logik befasst sich mit Wahrheitswerten und den darauf zugrunde liegenden Operationen. Als erstes möchte ich die Vergleichsoperationen illustrieren (LE = Linkes Element):

$<$	kleiner	LE kleiner als Rechtes?
$\leq$	kleiner gleich	LE kleiner oder gleich Rechtes?
$==$	Vergleich	LE gleicher Wert wie rechts?
$\geq$	größer gleich	LE größer oder gleich Rechtes?
$>$	größer	LE größer als Rechtes?

All diese Operationen geben einen boolschen Werte (wahr oder falsch) zurück. Aufgrund dieser Entscheidung kann dann der auszuführende Codeblock mithilfe einer bedingte Anweisung durchgeführt werden. Wichtig ist die Unterscheidung von = (Zuweisung) und == (Größenvergleich).

Größenvergleich ist für Nichtprogrammierer unter Identität besser verständlich, jedoch kann sich Gewichtsvergleich und Identität unterscheiden (zB in python, wo Elemente ident sind, wenn sie am selben Ort gespeichert werden, aber gleich, wenn sie die selben Größe besitzen.

Der andere Bereich ist die boolsche Algebra. Beispielweise ist die Anweisung "Nehme einen Apfel und eine Banane" nur dann erfüllt (wahr), wenn sowohl Apfel als auch Banane genommen wurde.

Für ODER gilt beispielweise...

Ereignis 1	Ereignis 2	Ereignis 1 ODER Ereignis 2
nicht e.	nicht e.	falsch
nicht e.	erfüllt	wahr
erfüllt	nicht e.	wahr
erfüllt	erfüllt	wahr

In der rechten Spalte können wir die Werte wahr, wahr, wahr und falsch lesen. Mit dem Alphabet 0 (n. e.) und 1 (erfüllt) bedeutet das 1110. Und mit diesem System sind die Ergebnisse der folgenden Tabelle in der linken Spalte notiert:

4.3.4 Prozeduren und Funktionen

Befehle wiederholen sich unweigerlich, wenn die Probleme etwas komplexer werden. Zum Beispiel nehmen ein Drittel der Rechenzeit eines Computers Sortieralgorithmen in Anspruch. Programmierer würden viel Zeit vergeuden, wenn sie diese Algorithmen bzw. ganz allgemein Codeblöcke mehrfach notieren müssten. Man teilt deshalb Codeblöcke in Einheiten auf. Diese Einheiten spricht man mit einem Namen an. Da ein Codeblock oft andere Vorgaben verwendet (zB die zu sortierende Liste), können Parameter verwendet werden. Funktionen sind Codeblöcke, die

Antworten	Bezeichnung
0000	falsch
1000	NOR
0100	Inhibition aus E_2
0010	Inhibition aus E_1
0001	AND
1100	Negation aus E_1
0110	XOR, Kontravalenz, Antivalenz
0101	E_2
1010	Negation aus E_2
0011	E_1
1110	NAND
0111	ODER
1111	wahr

mit einem Namen angesprochen werden und der Parameter übergeben werden. Als Resultat der Berechnungen wird ein Rückgabeparameter wieder retourniert.

Der Unterschied zwischen Funktionen und Prozeduren: Funktionen geben diesen Parameter zurück; Prozeduren nicht. Wozu braucht man einen Codeblock, der kein Ergebnis besitzt und somit nichts errechnet? Falsch, es gibt schon ein Ergebnis, nur muss darüber kein Resultat bekannt sein. zB könnte ich in einer Prozedur notieren, dass die graphische Oberfläche immer die Farbe Orange verwenden soll. Die GUI ("Graphical User Interface") sagt jetzt nicht, ob diese Operation erfolgreich war. Also kann auch die Prozedur keine näheren Angaben darüber retournieren und gibt keinen Wert zurück. Ein anderes Beispiel: Das gesamte Programm verändert die ganze Zeit nur eine Variable. Diese wird als "global" notiert, damit alle Funktionen und Prozeduren (und weitere Komponenten) darauf zurückgreifen können. Die Prozedur führt jetzt einen Algorithmus aus (zb $2 \cdot x$). Da diese Variable sowieso global behandelt wird, wird die Veränderung auch global wirksam und die Variable ist geändert worden. Die Prozedur braucht jetzt keine Variable verändern. Delphi ist die bekannteste Sprache, die diese strikte Unterscheidung führt. Andere Programmiersprachen machen keinen Unterschied. Entweder wird ein "return" (retournieren des Rückgabeparameters) ausgeführt (dann ist es eine Funktion) oder eben nicht (Prozedur).

```
# function

a = 5
def function(x):
    return x * 2

a = function(a)
print a

# procedure
b = 5
def procedure():
    global b
    b = b * 2

procedure()
```

```
print b

# this program outputs
# 10
# 10
```

Die Definition von Funktion stimmt mit der von der Mathematik überein.

4.3.5 GOTO

Wir möchten also Codeblöcke definieren, auf die wir jederzeit wieder zugreifen möchten. In modernen Sprachen werden diese Blöcke mit Namen angesprochen und Sonderzeichen zeigen welchen Teil der Codeblock umfasst. Das folgende Programm (Implementierung eines Algorithmus' in einer Programmiersprache) zeigt dies anhand eines Beispiels:

```
<?php
  function do_something() {
      echo 'Hello World!';
  }

  do_something();
  do_something();
  do_something();
  do_something();
?>
```

(PHP verwendet immer das Schlüsselwort *function* wobei es sich hier um eine Prozedur handelt)

Das Programm gibt viermal "Hello World!" aus. In dieser Programmiersprache (PHP) wird wie in C die geschwungene Klammer genutzt {}. Aber die Frage lautet: Wie wird dies im Computer selbst gehandhabt? Der Interpreter/Compiler übersetzt es ja in Maschinensprache. Wie schaut es dort aus? Die Antwort findet sich in antiken Programmiersprachen und lautet in etwa JMP oder GOTO.

Diese beiden Begriffe sind Schlüsselelemente mit denen innerhalb des Programms gesprungen werden kann. Das folgende Programm hat den selben Zweck wie das Programm oben. Ich habe nicht die "richtige" PHP Kontrollstruktur "goto" verwendet, sondern so angepasst, dass sie besser verständlich ist. Das folgende Programm ist also kein funktionierendre Quelltext (siehe Sektion 23).

```
1   i = 0
2   print 'Hello World!'
3   if (i < 4) {
4     i = i + 1;
5     goto line 2;
6   }
```

PASCAL ist bekannt für sein GOTO. GOTO kann entweder eine Zeilennummer ansprechen (siehe Beispiel oben) oder ein Label (wie in PHP eigentlich vorgesehen). Das GOTO-Statement ist umstritten, weil es den Quelltext unübersichtlich machen kann. Linus Torvalds und Linux-Entwickler sprechen sich aber für die Verwendung aus. Dementsprechend finden sich im Linux-Kernel eine Menge dieser Befehle.

4.4 in-place

Im Zuge der Sortieralgorithmen und in Zusammenhang mit Prozeduren (vs. Funktionen) möchte ich in-place ansprechen. Bei in-place-Funktionen handelt es sich um Funktionen, die den übergebenen Wert wirklich verändern und nicht bloß Werte (bei Sortieralgorithmen: eine geordnete Liste) zurückgeben. Das folgende Beispiel sollte es demonstrieren:

```
1   liste = [1, 2, 4, 3]
2
3   # out-of-place
4   liste = sorted(liste)
5
6   # in-place
7   sort(liste)
8
9   # anyway, it's [1, 2, 3, 4]
10  print liste
```

Bei out-of-place wird das Ergebnis empfangen und in die Variable a geschrieben. Bei in-place wird die Eingabe direkt verändert. Damit entspricht die Funktion *sort* einer Prozedur und *sorted* einer Funktion. python organisiert sich allgemein so, dass Funktionen mit der Endung *-ed* out-of-place arbeiten und die ohne Endung in-place.

4.5 Pseudocode

Bei Pseudocode handelt es sich um eine programmiersprachenunabhängige Notation. Für Kommandos werden keine spezifischen Namen von Programmiersprachen verwendet, sondern der naheliegendste Begriff, der die Aktion beschreibt. Für Zuweisungen wird das allgemein übliche Gleichzeichen = oder ein Pfeil → verwendet.

Die Pseudocodes können so auch gelesen werden ohne, dass man die genaue Schlüsselwörter oder syntaktischen Elemente einer speziellen Sprache verstehen muss. Daher ist Pseudocode für Seiten wie Wikipedia geeignet, wo versucht wird, Algorithmen allgemein zu erklären ohne auf Programmiersprachen einzugehen.

4.6 Iteration

Unter Iteration versteht man das Durchwandern eines Containerelements, welches ich bereits in der Sektion 19 angesprochen habe. Wie Iteration in einer Programmiersprache notiert wird, ist spezifisch. Jedoch steht es in jeder Sprache zur Verfügung. Oftmals sind die Elemente eines Containers mit Indizes ansprechbar. Dabei kann man die Indizes durch eine Laufvariable bei for-Schleifen erzeugen.

4.7 Rekursion

Bei Rekursion handelt es sich um Codeblöcke, die sich selbst aufrufen. Das Adjektiv heißt rekursiv und ist das Gegenteil von iterativ. Meine persönliche Lieblingsdefinition ist die aus einem

Wörterbuch:

> Re-kur-sion (-en/-en) siehe *Rekursion*

Man beginnt also Rekursion nachzuschlagen und in weiterer Folge nochmals. Und nochmals und nochmals bis man in einer Endlosschleife endet. Eine Rekursion ist eine Funktion, die sich selbst aufruft. Oben haben wir ein Beispiel wo wir eine Endlosschleife erhalten, jedoch haben wir genauso begrenzte Schleifen, wie zB bei der rekursiven Implementierung des euklidischen Algorithmus'.

```
1 def euklid(a, b):
2     if b == 0:
3         return a
4     else:
5         return euklid(b, a % b)
```

Ein weiterer und wesentlich komplexerer Algorithmus ist PageRank. PageRank beurteilt Webseiten auf Basis der Links und ist das ursprüngliche Konzept der Suchmaschine Google.

4.7.1 rekursiv vs. iterativ

Jedes Problem, welches rekursiv gelöst werden kann, kann auch iterativ gelöst werden. Nicht jedoch umgekehrt. Typischerweise findet man in der iterativen Implementierung eines ursprünglich rekursiven Algorithmus' eine while-Schleife.

4.8 Programmiersprachen

Was ist jetzt eine Programmiersprache? Eine Programmiersprache ist im Grunde eine Notation, die in die (Maschine)sprache des Computers übersetzt werden und damit Berechnungen durchgeführt werden können. Jeder Mensch kann es sich vorstellen, dass es komplizierter ist Einsen und Nullen zu schreiben als mit Schlüsselwörter umzugehen. Einsen und Nullen sind wesentlich fehleranfälliger (eine Null zu viel kann den Computer schon abstürzen lassen) und enthält oft Wiederholungen (DRY – "Don't repeat yourself"). Solche Probleme versuchen Programmiersprachen zu beheben. Ich nenne wie folgt ein paar:

- C, Objective-C, C++
- C#, Java
- python, ruby
- bash, zsh
- PHP, Perl
- Fortran
- BASIC
- PASCAL

- LISP
- Haskell

Alle Programmiersprachen lassen sich jetzt nach Kriterien erfassen (zB Turing-Vollständigkeit), nach Paradigmen gliedern (Programmierparadigmen) und Phasen der Entwicklung zuordnen (Generationen).

4.8.1 Turing-Vollständigkeit

Jeder kann seine eigene Programmiersprache entwickeln. In etwa die Hälfte (nach eigener Schätzung) der Programmiersprachen ist aus Anlass für Firmenzwecke entwickelt worden. Die Firma erschließt einen neuen Arbeitsbereich und benötigt hierfür eine Sprache, in der sich die behandelten Probleme besser formulieren lassen. Entwickelt wurden die meisten Programmiersprachen von Programmierern aber aufgrund des philologischen Hintergrunds finden sich genauso Linguisten als Entwickler.

Ok... du entwickelst deine eigene Programmiersprache. Sagen wir du möchtest bedingte Anweisungen notieren können, aber Schleifen brauchst du nicht für deinen Anwendungsbereich. Mit dem Fehlen von Schleifen kann man einige Probleme nicht mehr lösen. Damit gilt nicht mehr der Standard, dass mathematische und andere Probleme (wie Turing an der Turingmaschine illustrierte) lösbar sind. Turing hat hierfür den Begriff der Turing-Vollständigkeit eingeführt: Ist ein Problem (genauer: welches auf einer Turingmaschine lösbar wäre) nicht in einer Programmiersprache formalisierbar, so ist die Sprache nicht turing-vollständig. Ist ein Problem nicht auf einem Computer bearbeitbar, so ist der Computer nicht turing-vollständig.

(Um das Kriterium der Turing-Vollständigkeit zu erfüllen, ist es nicht notwendig einen unendlichen Speicher zur Verfügung zu stellen, wie es bei der Turingmaschine – mit dem unendlichen Magnetband – definiert wurde)

4.9 Programmierparadigmen, Generationen

Es gibt verschiedene Lösungsansätze an Probleme. Dies haben wir oben bei den besprochenen Problemen gesehen. Genauso geht es bei Programmiersprachen nicht nur darum die Probleme zu lösen, sondern auch darum, dass ein Programmierer sie effizient lösen kann. Ich erkläre hier einige Programmierparadigmen und schauen uns nachher an, wie man sie in Generationen einteilen kann.

imperativ Die Befehle werden schrittweise abgearbeitet, wobei besonderer Wert auf die klare Definition der einzelnen Schritte gelegt wird (zB C, Pascal, Fortran). Typisch für solche Sprachen sind häufige Anwendung von bedingten Anweisungen und globalen Variablen

logisch Das Paradigma basiert auf der Idee der mathematischen Logik. Klassisches Beispiel ist die Verwandtschaft. Der Vater von Daniel ist Matthias. Wer ist Matthias Sohn? (zB Prolog)

iterativ rekursive Ausdrücke sind nicht erlaubt, um die Effizienz zu erhöhen (zB ursprüngliches PASCAL)

prozedural Im Zentrum stehen Prozeduren. Ein Programm wird in kleine Prozeduren aufgeteilt, die globale Variablen verarbeiten bzw. anderen Komponenten (GUI, anderen Programmbibliotheken) (zB Fortran, ALGOL, COBOL)

funktional Im Zentrum stehen Funktionen. Diese werden am Programmanfang notiert und eine Hauptfunktion ruft dann einzelne Funktionen auf, die zur Programmabarbeitung notwendig sind. Dadurch wird beispielweise der Programmablauf stark verschleiert. Typisch sind auch stark reduzierte Ausdrücke, die eine leichte Verarbeitung von Containerelementen erlaubt (zB Haskell)

modular Die Komplexität von Programmen wird minimiert indem man Programme in Module unterteilt, die verschiedene Teilprobleme lösen

deklarativ Es wird nicht der Algorithmus spezifiziert, sondern das Resultat. Es wird also keine Anweisung gegeben, wie eine Liste geordnet werden soll, sondern wie das Ergebnis aussieht (das jetzt darauffolgende Element muss größer als das vorhergehende sein) (zB Haskell, SQL, XSLT)

strukturiert basierend auf prozedural, jedoch unterscheidet man nur Sequenzen (Codeblöcke), Auswahlen (bedingte Anweisungen) und Wiederholungen (Schleifen)

objektorientiert Daten werden als Einheiten (Objekte) aufgefasst, die eine Identität (Klassennahmen), Verhalten (Methoden, functions) und Eigenschaften (properties). Die Objekte werden als Klassen definiert und Instanzen sind laufende Kopien von Klasse mit spezifischen Eigenschaften. Ermöglicht wird dadurch Vererbung (Kopieren von Methoden einer höheren Klasse), Polymorphismus (Abhebung von Datentypen-Basis) und Kaspelung (Daten liegen im privaten oder öffentlichen Bereich vor)

nebenläufig, parallel Progammkomponenten werden so weit es geht in Teilprobleme untergegliedert, damit sie parallel abgearbeitet werden können und die Laufzeit minimiert wird.

hybrid Vereint mehrere der verschiedenen Programmierparadigmen

Um einen Überblick zu geben, verzichte ich auf die genaue Definition und richtige zeitliche Zuordnung der Programmierparadigmen,aber möchte dadurch eine klareren Überblick geben (heuristisch vollführt sozusagen):

> Programme wurden ursprünglich schrittweise angegeben. Beispielweise soll zu einer Zahl der Wert Eins addiert werden und dann durch 2 dividiert werden (imperativ). Man führt jedoch bedingte Anweisungen ein, um auf Ereignisse reagieren zu können und führt Datentypen ein, damit man mit unterschiedlichen Daten arbeiten kann. Man möchte sich von der Mathematik entfernen, behält jedoch die logischen Aspekte bei, da sie unverzichtbar sind bei der Lösung von Problemen (logisch).
>
> Man hat das Problem des DRY – Wiederholung von Befehlssequenzen. Man möchte dem Programmierer die Arbeit erleichtern und teilt Programme in kleinere Programme auf (prozedural). Jedoch gibt es noch einige Probleme, wenn Variablen von Komponenten verändert werden, die von ganz anderen Bibliotheken stammen. Man weist jedem kleineren Programm einen eigenen Namensraum zu (funktional). Die neue Idee auf Basis der funktionalen Programmierung war die Rekursion. Einige Programmiersprachen finden diese Idee toll und implementieren sie, aber einige wehren sich jedoch dagegen, da sie oft langsamer arbeiten (iterativ).
>
> Software ist komplex. Das bedeutet, dass ein kleine Gruppe von Programmierern nicht mehr ausreicht, um den Quelltext eines großen Produkts verwalten kann. Als Konsequenz teilt man das Programm in mehrere Einheiten. Funktionen und Prozeduren sind hierfür ein Ansatz, aber sollten mehr diesen dateiübergreifenden Ansatz

besitzen. Man beginnt also damit Module zu definieren, die über Schlüsselwörter importiert werden (modular).

Die Softwareentwicklung wird aber auch vielfältig. Man ist aufgeschlossen gegenüber neuen Ideen. So besteht darin eine Idee Programme zu notieren, wo nur das Resultat definiert wird, der Weg weitgehendst nicht (deklarativ). Man besitzt jedoch das Problem der Effizienz. Die Programmiersprache bildet dadurch nicht mehr einen schönes Mittelweg zwischen Optimierung und Menschlichkeit für den Programmierer. Man versucht sich also wieder zurück zur Basis zu orientieren und die Sprache leicht verständlich zu machen (strukturiert) und Arbeiten parallel ablaufen zu lassen (nebenläufig).

Doch in den 60ern ist eine Paradigma aufgekommen, welches die bisherigen Problemansätze (Komplexität des Quelltexts, geteiltes Arbeiten, Effizienz, No-DRY) sauber in sich löst: Objektorientierung. Es handelt sich um einen völlig anderen Ansatz an Probleme heranzugehen.

Ok... ein netter Aufsatz, aber wesentlich inhaltlicher korrekter sind die Generationen nach denen Programmiersprachen geordnet werden[1]:

G	Sprachenart	Beispiel	Abstraktion
1	Maschinen	Maschinencode	Binäre Befehle
2	Assembler	Assembler-Code	Symbolische Befehle
3	prozedular	FORTRAN, COBOL	Hardwareunabhängig
3+	funktional, objektorientiert	PASCAL, C, C++, Java, C#	Strukturiert, OOP
4	deklarativ	LISP, PROLOG, SQL	Transaktions-orientiert
5	?		

Die Programmiersprachen ab G3 werden als "Hochsprachen" bezeichnet.

Was wir hier quasi neu haben: Maschinencode. Maschinencode ist nichts weiter als das, was Interpreter / Compiler von Programmiersprachen übersetzt. Man schreibt also einen Quelltext in C und jener wird in Maschinencode übersetzt. Was sind Interpreter und Compiler?

4.10 Interpreter vs. Compiler

Interpreter und Compiler haben die Aufgabe Quelltexte in Maschinensprache zu übersetzen und damit ausführbar zu machen. Der Unterschied liegt darin, dass Compiler das Programm translatieren – genauer: kompilieren – und dadurch eine fertige Datei speichern, die abgespeichert wird. Diese Datei ist Maschinencode und jederzeit ausführbar. Somit ist sie auch plattformabhängig ("Plattform" bezeichnet ein Betriebsystem mit all seinen Softwarekomponenten). Die kompilierte Datei muss nun ausgeführt werden, damit das Ergebnis sichtbar ist. Das Gegenstück sind Interpreter, wo der Quelltext in Echtzeit abgearbeitet wird und die Ausgabe direkt erfolgt. Es gibt keine kompilierte Dateien.

Natürlich gibt es wieder Mischformen. So gibt es JIT ("Just in time") Compiler, die Bytecode erstellen. Dieser Bytecode ist eine plattformunabhängige, fast kompilierte Datei. Java ist hierfür das bekannteste Beispiel. Bei python sieht man es ebenso: wird ein Programm auf der Konsole mit *python program.py* gestartet, so erzeugt python eine Datei *program.pyc*, die Bytecode enthält.

[1] zitiert aus "Grundlagen der Informatik" von Herold, Lurz und Wohlrab, S. 143

4.11 Eine Sprache im Detail: python

python hat imperative (ein Befehl pro Zeile und Zeile für Zeile abarbeiten), funktionale (List Comprehension, def), prozedurale (def, global), modulare (import), strukturierte (Einrückung, reduzierte Schleifenschlüsselnotation) und vor allem objektorientierte (alles ist ein Objekt, class) Aspekte. python lässt sich durch entsprechende Bibliotheken leicht zu einer logischen und nebenläufigen Sprache weiterentwickeln. Es handelt sich jedenfalls um eine hybride (der Begriff wird gerne vermieden, weil es ursprünglich die prozedural-objektorientierte Kombination bezeichnete), multiparadigmatische Skriptsprache. Sie ist turing-vollständig und entstammt der Generation G3+.

Kapitel 5

Dokument und Lernstoff

5.1 Status and Copyleft

5.1.1 Copyleft

Use this document as you would like to.

5.1.2 Status

Version alpha-overbuggy
not proofread!

Die Algorithmen sind in PHP, python und C geschrieben, da ich einmal an der Stelle mit PHP die C-Syntax brauchte, wobei ich nicht C selbst verwenden wollte (wegen main()), python syntaktisch wunderschön finde und C performant arbeitet. Mit PHP und python habe ich selbst am meisten Erfahrung.

Weitere (nicht behandelte) Stichwörter:

- Komplexitätsklassen (weitere und im Vergleich)
- Registermaschinen (und weitere Rechenautomaten)
- NP-Schwere
- Lambda-Kalkül
- Datentypen allgemein
- Dynamik (bez. Datentypen) von Programmiersprachen
- Unterschied Programmier- und Skriptsprachen
- Bitweise Berechnungen (Bit-Shift, ...)
- Türme von Hanoi
- Meta-Heuristik

- Nearest Neighbour Heuristik
- Greedy-Algorithmen
- Pseudozufallszahlen
- A* Algorithmus
- Rekursiv erzeugte Graphiken

5.2 Fragen

- Definiere den Begriff Algorithmus allgemein und gehe danach auf Turings Definition ein.
  - schrittweise Anleitung zur Lösung eines Problems
  - Chwarizmi, Babbage (Bernoulli-Zahlen), Turing, Church, ...
  - Finitheit
  - Ausführbarkeit
  - Platzkomplexität, dynamische Finitheit
  - Terminierung, Zeitkomplexität
  - Determiniertheit
  - Determinismus
- Zeige einfache Algorithmen und ihre visuellen Darstellungen
  - Identische Abbildung
  - Division
  - Struktogramm (Nassi-Shneiderdiagramm, Blöcke)
  - Flussdiagramm (Pfeile)
- Zeige einige Probleme auf und definiere sie bezüglich ihrer Lösbarkeit
  - Problem des Handlungsreisenden (unlösbar)
  - Speisende Philosophen (bedingt lösbar)
  - Rucksackproblem (lösbar)
  - Damenproblem (lösbar)
  - Landkartenfärbungsproblem (lösbar)
- Definiere allgemeine Lösungsansätze an diese Probleme
  - Heuristik
  - Backtracking
  - Divide and conquer
- Erkläre die Grundelemente der beiden Wissenschaften aus der theoretischen Informatik
  - siehe oben (Finitheit, ...)

  - Landau-Symbole
  - Komplexitätsklasse
- Erkläre die Turingmaschine und ihre Bedeutung
  - Aufbau
  - Anwendung
  - Halteproblem
- Analysiere den Algorithmus MergeSort nach den genannten Kriterien
  - Funktionsweise
  - Stabilität
  - rekursiv / iterativ?
  - (best, worst, average case) Zeitkomplexität
  - Raumkomplexität
- Implementiere den euklidischen Algorithmus auswendig
  - alt, iterativ
  - alt, rekursiv
  - modern, iteratv
  - modern, rekursiv
- Erkläre Grundelemente von Programmiersprachen
  - Bedingte Anweisungen
  - Schleifen (for, while, Container-Iteration)
  - Logik (Vergleichsoperatoren, Boolsche Algebra)
  - Prozeduren vs. Funktionen
  - GOTO
- Erkläre die Begriffe Turing-Vollständigkeit, Programmierparadigma und Generation in Bezug auf Programmiersprachen und teile sie historisch zu
  - imperativ, logisch, iterativ, prozedural, funktional, modular, deklarativ, strukturiert, objektorientiert, nebenläufig, hybrid
  - 1, 2, 3, 3+, 4. Begriff Hochsprachen
  - FORTRAN als erste Hochsprache Sprache
  - PASCAL, BASIC, C, C++, python, Java, C# folgend
  - LISP, PROLOG, SQL
- Nenne Beispiele für Programmiersprachen und gehe näher auf die besprochenen Kriterien ein, die an Programmierer und Maschine gestellt werden
  - python wurde im Dokument besprochen (Interpreter, Compiler, erlaubte Paradigmen, Generation, Turing-Vollständigkeit)

5.3 Fragen

- Wie nennt man das? Dreieckstausch?
- n*log(n) liegt zwischen n und n^2

5.4 Vorgehensweise bei Matura

- Immer Problemaufriss machen!
- Struktogramme, Flussdiagramme vorbereiten
- Möglichst Überschriften aus Dokument verwenden
- Graphiken, Übersichten vorbereiten
- Algorithmen genau durchdenken