

# Die Ordnung von Algorithmen

In meinen Anfangstagen des Programmierens (als Student am Physikinstitut) wollte ich einmal einen physikalischen Effekt berechnen, der mich interessierte. Dazu musste ich ein lineares Gleichungssystem lösen, das typischerweise um die 40 Variable hatte. Ich klemmte mich hinter den Computer (ein altes aber flottes Ding mit CP/M und Fortran). Es klappte, die Berechnung dauerte allerdings eine Stunde. Ich war sehr stolz und feilte am Programm, um es noch etwas schneller zu machen. Meine Begeisterung war groß, doch die Langsamkeit der Gleichungslösung betrückte mich. Allerdings wusste ich wenigstens woran es lag – Gleichungen zu lösen ist eben mühsam, wie wir in der Mathematikvorlesung gelernt hatten. Ich überlegte schon, ob ich nicht den Versuch machen sollte, diesen Teil in Maschinensprache zu schreiben....

Mein Leid klagte ich einem Studienkollegen, der sich in Assembler gut auskannte. Ich schilderte ihm das Problem und er versprach, es sich anzusehen. Allerdings war er an Mathematik nicht allzu interessiert und hatte in der Vorlesung die Determinanten nur halb mitbekommen. Er holte sich in der Bibliothek ein Buch um nachzulesen.

Wenig später kam er und sagte, er hätte jetzt eine Version, die deutlich schneller lief, ich würde mich aber nicht darüber freuen. Ich machte mich schon auf schlimmste Assembler-Prozeduren gefasst, als er mir sein Werk zeigte: Auf einem uralten Apple-2 in Basic. Und das Programm lief wirklich flotter!

Was war passiert: Er hatte im Buch einen passenden Algorithmus gefunden, und um ihn zu testen und eine Vorlage für die Assemblerumsetzung zu haben, tippte er ihn rasch in Basic ein. Aber wieso war das Programm so schnell gegenüber meinem Fortran-Programm?

Wir testeten mit unterschiedlicher Gleichungsanzahl und maßen die Zeit:

- Bis zur Größe 8 hatte mein Programm die Nase vorn.
- Bei 9 Gleichungen waren sie etwa gleich schnell
- Und ab dann wurde mein HighTech-Fortran zur Schnecke und das 'mickrige' AppleBasic zog unaufhaltbar davon.

Es konnte also nicht an der Geschwindigkeit der Computer liegen, hier musste etwas viel grundlegenderes passiert sein: wir beide hatten doch unterschiedliche Programme geschrieben. Ich nach mathematischem Vorlesungswissen, mein Kollege nach einem Buch über mathematische Methoden der Physik.

Unser Professor gab uns dann den Tip, das sogenannte 'Zeitverhalten' unserer Programme zu untersuchen, und damit konnten wir nach einem Kurzbesuch in der Bibliothek dem Problem auf den Grund kommen:

Mein Programm benötigte für die doppelte Anzahl von Gleichungen 8 mal so lange, für die dreifache 27 mal. Sein Programm allerdings für die doppelte Anzahl nur 4 mal so lang, für die dreifache nur 9 mal so lang!

Als Tabelle der Zeitbedarf, wenn wir (genähert) annehmen, dass 10 Gleichungen 1 Minute brauchen:

Anzahl der Gleichungen	10	20	50	100	200
mein Programm	1 Min	8 Min	2 Stunden	17 Stunden	>5 Tage
mein Kollege	1 Min	4 Min	25 Min	1½ Stunden	>6 Stunden

Wir können zur Beschreibung diese Verhaltens die sogenannte **O-Notation** der Mathematiker verwenden. (Achtung: Mathematiker kennen auch eine o-Notation, die etwas anders ist!) Informatiker sprechen auch gern von der **Komplexität einer Aufgabe**.

Mein Programm braucht für  $n$  Gleichungen eine Laufzeit proportional  $n^3$ , ist also  $O(n^3)$ .  
 Sein Programm braucht für  $n$  Gleichungen eine Laufzeit proportional  $n^2$ , ist also  $O(n^2)$ .  
 Man sagt 'der Algorithmus ist von Ordnung  $n^2$  bzw.  $n^3$ '.

Und es zeigte sich: ein langsamer Computer mit einem rascheren Algorithmus ist besser als ein schneller Computer mit einem schlecht gewählten Algorithmus – wenn es um eine große Zahl von Gleichungen geht.

Typische Beispiele:

$O(1)$ :	Berechnung der Summe der ersten $n$ Zahlen mit $n*(n+1)/2$ in konstanter Zeit
$O(n)$ :	Berechnung dieser Summe durch Aufaddieren der Summanden
$O(n^2)$ :	einfache Sortieralgorithmen
$O(n \log n)$ :	Sortieralgorithmen mit 'divide and conquer'
$O(n^3)$ :	einfache Matrizenmultiplikation
$O(n!)$ :	'schwere' Probleme, wie die Rundfahrt des Handlungsreisenden

Um die Komplexität herum ist eine ganze Teildisziplin der Informatik entstanden. Sie hat etwa auch die Aufgabe herauszufinden, was die kleinstmögliche Komplexität für eine gestellte Aufgabe ist ('optimale Algorithmen').

Leider ist diese optimale Ordnung eines Algorithmus in vielen Fällen bis heute nicht genau berechenbar (Shell-Sort), oder es ist gar kein Algorithmus bekannt, der so effizient arbeitet.

Es warten noch genug Herausforderungen auf mathematisch versierte Informatiker! Es zeigt sich nämlich, dass viele 'akademisch' anmutende Informatik-Probleme durchaus praktische Anwendungen in der Praxis haben. So ist das Problem des Handlungsreisenden ganz genau das Problem des Roboter-LötKolbens beim Fertigstellen einer Platine – und hier ist jeder Zentimeter weniger Weg eine Zeitersparnis, die sich in höheren Produktionsraten niederschlägt und am Ende in klingender Münze messbar ist. Nicht umsonst hat etwa IBM eine riesige (und mit hochkarätigen Wissenschaftlern besetzte) theoretische mathematische Grundlagenforschungsabteilung.

## Ein Beispiel - Anagramme

Anagramme sind Buchstabenvertauschungen eines Wortes, die wiederum ein sinnvolles Wort ergeben. LAUF – FAUL – FLAU, FERIEN – REIFEN, NEGER – REGEN sind solche Anagramme.

**Die Aufgabe:**

**Du bekommst eine Liste von Worten (in Großbuchstaben) und sollst alle darin befindlichen Anagramme herausfinden.**

Überlege Dir selbst einen Algorithmus, bevor Du weiterliest!

**Lösung 1:**

Nimm das erste Wort. Bestimme alle seine Anagramme (Funktion *permutiere*). Wenn eines davon in den übrigen Worten vorkommt, zeig die Werte an. Dann mach mit dem zweiten Wort weiter, und so fort.

Das Programm funktioniert sicher, aber: Wenn etwa das erste Wort RIESENSCHLAMPEREI heißt, so sind aus ihm alleine fast  $1.85 \times 10^{12}$  Vertauschungen zu untersuchen. Auch der schnellste Computer des Universums käme da nicht mehr mit! Dieser Algorithmus wird uns nicht weiterhelfen. Seine Ordnung ist astronomisch...

**Lösung 2:**

Nimm das erste Wort. Untersuche, ob eines der folgenden Worte ein Anagramm dieses Wortes ist (Funktion *hatselbebuchstaben*). Wenn ja, gib das Paar aus. Dann mach mit dem zweiten Wort weiter, und so fort.

Sieht besser aus.  $n$  Worte in der Liste. Jedes wird den auf seine Position folgenden Worten verglichen, diese restliche Anzahl hängt auch linear mit  $n$  zusammen. Wir haben einen Algorithmus der Ordnung  $n^2$  gefunden. Wenn die Liste 100 Elemente hat, super. Aber wenn sie 10000 Worte enthält?

**Lösung 3:**

Bereite eine zusätzliche Liste vor: Eine Liste, wo jedes Wort einem Code zugeordnet ist (Funktion *codiere*), der nur von den Buchstaben des Wortes abhängt, nicht aber von ihrer Reihenfolge. Nun suche alle Schlüssel, die mehr als einen Eintrag enthalten und gib ihre Werte aus (Funktion *ausgabe*).

Der Vorbereitungsteil hat  $O(n)$ , der Ausgabeteil ebenfalls  $O(n)$  – wie haben einen Algorithmus der Komplexität  $n$  gefunden! Und besser als linear kann es nicht gehen, da jedes Wort einmal berücksichtigt werden muss.

**Umsetzung in Python:**

Der Rest der Programmiererwelt beginnt nun eifrig über die Funktion *codiere* nachzudenken und informiert sich über Hashing-Tables. Wir sind in Python fein raus – das mächtige Werkzeug der Directories kann bereits alles, was wir brauchen.

Als Schlüssel wählen wir einfach das nach den Buchstaben sortierte Wort! Dann zählt jeder Buchstabe, aber die Reihenfolgeinformation ist verschwunden.

```
Aus
[FAUL, HEXE, LAUF]
wird dann ein Directory
{AFLU: [FAUL,LAUF], EEHX: HEXE}
```

und jetzt durchlaufen wir alle Schlüssel (keys) und geben die zugehörige Liste aus, wenn sie mehr als 1 Element hat.

```
for x in verzeichnis.values():
    if len(x)>2: print x
```

Und die Erzeugung der Schlüssel? Der alte Trick mit der Stringliste, da wir natürlich die eingebaute Sortieroutine verwenden. Folgende Zeilen werden einfach für jedes Wort in der Eingabeliste ausgeführt:

```
sortwort = list(wort)
sortwort.sort()
key = sortwort
if verzeichnis.has_key(key):
    verzeichnis[key].append(wort)
else:
    verzeichnis[key]=wort
```

Und schon sind wir fertig!