4.3 Algorithmenanalyse

4.3.1 Zuverlässigkeit

Die wichtigste Forderung an ein Programm ist nicht Schnelligkeit, sondern dass es tut, was es soll. Dass auch jahrelang scheinbar zuverlässige Software plötzlich falsche Ergebnisse liefert, zeigen diese Fälle:

- **Ariane 5, 1996.** Vierzig Sekunden nach dem Start zerstörte sich die Rakete selbst. Ursache war eine aus der Ariane 4 übernommene Routine, die eine 64-Bit-Gleitkommazahl in eine 16-Bit-Ganzzahl umwandelte. Für die schnellere Ariane 5 war der Wert zu groß. Der Code funktionierte fehlerfrei — nur für die falsche Rakete.
- **Jahr-2000-Problem.** Um teuren Speicherplatz zu sparen, wurde die Jahreszahl jahrzehntelang zweistellig gespeichert. Datumsvergleiche konnten 1900 und 2000 nicht unterscheiden. Die weltweite Umstellung kostete geschätzt mehrere hundert Milliarden Dollar.
- **Boeing 737, Startabbruch bei 60 Knoten.** Das Startprogramm behandelte die Fälle $v < 60$ und $v > 60$ — nicht aber $v = 60$. Meldeten die Sensoren genau diesen Wert, erkannte das Programm einen undefinierten Zustand und brach ab.

Der letzte Fall ist der lehrreichste, weil er keine exotische Technik voraussetzt: Ein einziger vergessener Randfall genügt. Als Faustzahl rechnet man in der Softwaretechnik mit mindestens einem Fehler je 10 KByte Quelltext.

Definitionen. Der Nachweis, dass ein Informatiksystem zuverlässig arbeitet, heißt **Validierung**. Man unterscheidet dabei zwei Wege:

- Bei der **Verifikation** wird formal, also durch logische Herleitung bewiesen, dass der Algorithmus seine Bedingungen einhält.
- Beim **Testen** wird das Programm mit ausgewählten Daten erprobt, deren Ergebnis bekannt ist.

Ein Algorithmus heißt **korrekt**, wenn er seiner Spezifikation genügt: Zu allen Eingabedaten, die eine bestimmte Vorbedingung erfüllen, liefert er Ausgabedaten, die eine bestimmte Nachbedingung erfüllen.

Merke

Testen kann nur die Anwesenheit von Fehlern zeigen, nie ihre Abwesenheit (Edsger Dijkstra). Wer alle Eingaben durchprobieren wollte, müsste bei einem Programm mit zwei 32-Bit-Zahlen als Eingabe rund $1{,}8 \cdot 10^{19}$ Fälle prüfen.

Arbeitsauftrag 7

27. Erklären Sie den Unterschied zwischen Verifikation und Testen an Ihrem Sortierprogramm: Was ließe sich beweisen, was nur testen?
28. Formulieren Sie Vor- und Nachbedingung für ein Sortierprogramm möglichst genau. Achtung: „Die Ausgabe ist aufsteigend sortiert" allein genügt nicht — warum?
29. Welche Testfälle würden Sie für ein Sortierprogramm wählen? Nennen Sie mindestens fünf und begründen Sie jeden.

4.3.2 Effizienz und Größenordnung

Zwei Algorithmen heißen **äquivalent**, wenn sie gleichen Eingabedaten gleiche Ausgabedaten zuordnen. Von zwei äquivalenten Algorithmen ist derjenige **effizienter**, der mit weniger Laufzeit und/oder weniger Speicherplatz auskommt. Beides zugleich lässt sich meist nicht beliebig verbessern: Ab einer gewissen Schwelle wird Geschwindigkeit mit Speicher erkauft und umgekehrt.

Die Laufzeit in Sekunden zu messen, hilft beim Vergleich von Algorithmen wenig. Sie hängt vom Prozessor ab, vom Compiler, vom Cache, vom Betriebssystem und davon, was sonst noch läuft. Deshalb zählt man stattdessen **Elementarschritte** — bei Sortierverfahren die Vergleiche.

Bezeichnung. Ein Problem der Größe n ist ein Problem, dessen Eingabe aus n Elementen besteht. Die Zahl $T(n)$ der Rechenschritte, die ein Algorithmus dafür benötigt, heißt **Zeitkomplexität**.

Zur Ermittlung nimmt man drei Vereinfachungen vor:

1. Man betrachtet nur die **dominante Operation**, von der die Laufzeit wesentlich abhängt.
2. Man unterscheidet **besten**, **schlechtesten** und **mittleren Fall**.
3. Man lässt additive und multiplikative Konstanten weg und geht zur **Größenordnung** über.

Der dritte Punkt ist der wichtigste und beim ersten Lesen der befremdlichste. Er besagt: Ob ein Verfahren n^2 oder $\frac{n^2}{2} + 3n + 17$ Schritte braucht, ist für große n gleichgültig — entscheidend ist das Wachstum. Notiert wird das in der **Landau-Notation**:

- $\mathcal{O}(n)$ — **lineares** Wachstum. Doppelt so viele Daten, doppelte Laufzeit.
- $\mathcal{O}(n^2)$ — **quadratisches** Wachstum. Doppelt so viele Daten, vierfache Laufzeit.
- $\mathcal{O}(n \log n)$ — dazwischen, und für Sortierverfahren die entscheidende Klasse.
- $\mathcal{O}(2^n)$ — **exponentielles** Wachstum. Ein Element mehr verdoppelt die Laufzeit. Schon bei kleinen Eingaben praktisch unbrauchbar.

4.3.3 Unsere vier Verfahren im Vergleich

Die folgende Tabelle enthält gemessene Vergleichszahlen: Mittelwerte über 300 zufällige Felder. Die letzte Spalte ist mit den Formeln hochgerechnet.

Verfahren	**Zeitkomplexität**	$n = 20$	$n = 1000$	$n = 100\,000$ **(gerechnet)**
Selection Sort	$\mathcal{O}(n^2)$, genau $\frac{n(n-1)}{2}$	190	499 500	$5 \cdot 10^9$
Bubble Sort	$\mathcal{O}(n^2)$, etwa $\frac{n^2}{2}$	181	498 530	$5 \cdot 10^9$
Insertion Sort	$\mathcal{O}(n^2)$, im Mittel etwa $\frac{n^2}{4}$	113	250 315	$2{,}5 \cdot 10^9$
Quicksort	$\mathcal{O}(n \log n)$, etwa $1{,}4\, n \log_2 n$	71	10 947	$2{,}3 \cdot 10^6$

Der Sprung in der letzten Spalte ist der eigentliche Punkt dieses Abschnitts: Bei hunderttausend Elementen trennt Quicksort von den anderen drei Verfahren nicht ein Faktor 2 oder 10, sondern ein Faktor von über tausend — gegenüber Selection und Bubble Sort sogar mehr als zweitausend. Ein Rechner, der die quadratischen Verfahren in einer Stunde schafft, wäre mit Quicksort nach wenigen Sekunden fertig. Kein Hardwarekauf der Welt gleicht das aus — der Unterschied liegt im Algorithmus.

Wichtig ist dabei die Unterscheidung der Fälle:

Verfahren	bester Fall	mittlerer Fall	schlechtester Fall
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Bubble Sort	$\mathcal{O}(n)$ (sortiert)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion Sort	$\mathcal{O}(n)$ (sortiert)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$ (ungünstiges Pivot)

Quicksort ist also nicht in jedem Fall schnell: Wählt man stets das letzte Element als Pivot und ist das Feld bereits sortiert, zerlegt jede Runde das Feld in einen leeren und einen fast vollständigen Teil — das Verfahren wird quadratisch. In Python macht sich dieser Fall sogar doppelt bemerkbar: Die Rekursion wird dann n Ebenen tief, und ab etwa tausend Elementen bricht das Programm mit `RecursionError` ab, lange bevor die Laufzeit auffällt. Genau das haben Sie in den Aufgaben 21 und 25 beobachtet. Abhilfe schafft eine bessere Pivotwahl, etwa das mittlere von drei Elementen.

Merke

Kein allgemeines Sortierverfahren, das nur vergleicht, kommt mit weniger als $\mathcal{O}(n \log n)$ Vergleichen aus. Begründung: Bei n Elementen gibt es $n!$ mögliche Reihenfolgen. Jeder Vergleich hat zwei Ausgänge und kann die Zahl der in Frage kommenden Reihenfolgen also höchstens halbieren. Man braucht deshalb mindestens $\log_2(n!) \approx n \log_2 n$ Vergleiche, um eine einzige übrig zu lassen. Quicksort liegt im Mittel in dieser Größenordnung — besser geht es nicht.

Arbeitsauftrag 8

30. Ein Algorithmus braucht für $n = 1000$ genau 2 Sekunden. Wie lange braucht er für $n = 4000$, wenn er linear, quadratisch bzw. exponentiell wächst?
31. Begründen Sie mit der Formel $\frac{n(n-1)}{2}$, warum Selection Sort bei Verdoppelung der Elementzahl etwa viermal so viele Vergleiche braucht. Prüfen Sie es mit der Visualisierung nach.
32. Warum hat Bubble Sort einen besseren besten Fall als Selection Sort, obwohl beide im Mittel gleich viele Vergleiche brauchen?
33. Erweitern Sie Ihre Sortierprogramme um Zähler für Vergleiche und Vertauschungen. Sortieren Sie damit Felder der Größe 100, 200, 400 und 800 und tragen Sie die Werte in eine Tabelle ein. Bestätigen Sie das quadratische Wachstum.
34. Ein Verfahren mit $\mathcal{O}(2^n)$ schafft $n = 40$ in einer Stunde. Schätzen Sie die Laufzeit für $n = 60$. Kommentieren Sie das Ergebnis.
35. Diskutieren Sie: Für $n = 10$ ist Insertion Sort in der Praxis schneller als Quicksort, obwohl die Größenordnung schlechter ist. Wie passt das zusammen? (Stichwort: weggelassene Konstanten.)