Berechenbar, aber zu langsam oder zu teuer

Im vorhergehenden Abschnitt wurden Probleme vorgestellt, die überhaupt nicht und niemals gelöst werden können. Darüber muss man sich also nicht mehr den Kopf zerbrechen. In diesem Abschnitt geht es um Probleme, die zwar berechenbar sind, aber wegen des Laufzeitverhaltens der Algorithmen oder des Speicherbedarfs nur theoretisch berechenbar sind, aber auf einer realen Maschine praktisch niemals berechenbar sein werden.

Was ist ein langsames, was ein schnelles Programm?

Der Begriff der Zeitkomplexität eines Algorithmus' wurde 1960 von RABIN eingeführt. Sie ist eine arithmetische Funktion, die den Aufwand an Rechenzeit in Abhängigkeit vom Umfang des Problems (Anzahl der Eingabedaten, Ordnung einer Matrix, Grad eines Polynoms o.ä.) angibt. Man unterscheidet dabei zwischen dem Aufwand im Mittel (*average case*) und dem Aufwand im schlimmsten Fall (*worst case*). Meist wird der rechnerische Aufwand nur größenordnungsmäßig abgeschätzt. Um solche Größenordnungen von Funktionen auszudrücken, hat sich die sogenannte *groß-Oh*-Notation bewährt.

> Ein Algorithmus muss für die Lösung eines Problems in Abhängigkeit einer Größe n eine bestimmte Anzahl $A(n)$ an elementaren Operationen ausführen.
>
> Der Algorithmus hat dann die Zeitkomplexität $O(g(n))$, wenn es eine Konstante c und eine Funktion g gibt, so dass $A(n) \leq c \cdot g(n)$ für alle $n \in \mathbf{N}$ gibt.

Beispiele:

1. Mit sogenannten *Rupfblumen* wie Margeriten, Gänseblümchen u.a. lässt sich auf einfache Art und Weise der „Sie liebt mich! — Sie liebt mich nicht!"- Algorithmus durchführen, bei dem der worst case mit dem average case identisch ist und der Algorithmus immer anhält. Er hat in Abhängigkeit von der Anzahl n der Blütenblätter die Zeitkomplexität $O(n)$.

2. Wir suchen in einer Liste mit n Elementen nach einem bestimmten Element. Bei jedem Eintrag ist zu prüfen, ob der aktuelle Eintrag dem zu suchenden Element gleicht und ob das Listenende schon erreicht ist. Also ist in diesem Falle $A(n) = 2 \cdot n$.

   Dieser Algorithmus hat also die lineare Zeitkomplexität $O(n)$.

   Man beachte: Durch irgendwelche Programmiertricks könnte man den Zeitbedarf durchaus um 50% senken. Dies ändert aber nichts an der Zeitkomplexität, die hier einfach nur besagt, dass für die doppelte Anzahl an Elementen die doppelte Zeit benötigt wird.

3. Bei den einfachen Sortieralgorithmen *MinSort* und *Bubblesort* haben wir festgestellt, dass die Rechenzeit proportional zum Quadrat der Anzahl der zu sortierenden Elemente ist.

   Diese Algorithmen haben also die quadratische Zeitkomplexität $O(n^2)$.

4. Bei der binären Suche stellten wir fest, dass die Anzahl der Vergleiche proportional zu $\log_2 n$ ist.

   Dieser Algorithmus hat also eine logarithmische Zeitkomplexität $O(\log_2 n)$. Auf die Basis des Logarithmus kommt es dabei nicht an, da $\log_a b = \frac{\ln b}{\ln a}$. Der Unterschied ist wieder nur eine Konstante.

   Algorithmen mit dieser Zeitkomplexität sind sehr, sehr schnell.

5. Bei den anspruchsvolleren Sortieralgorithmen *Quicksort* und *Mergesort* kann man beweisen, dass im Mittel eine Zeitkomplexität $O(n \cdot \log n)$ besteht. Die Basis des Logarithmus ist unwichtig.

6. Für das Problem, für eine Zahl festzustellen, ob sie eine Primzahl ist, ist erst kürzlich ein Beweis von den indischen Mathematikern AGRAWAL, KAYAL und SAXENA vorgelegt worden, der besagt, dass das Problem im worst case mit $O(\log^{12}(n))$ und im average case mit $O(\log^6(n))$ gelöst werden kann.

7. Beim Problem der *Türme von Hanoi* haben wir festgestellt, dass zum Umsetzen von n Scheiben $2^n - 1$ Scheibenbewegungen nötig sind.

   Dieser Algorithmus hat also die exponentielle Zeitkomplexität $O(2^n)$.

   Die Basis der Exponentialfunktion ist wiederum unwichtig, weil alle anderen Basen sich nur durch einen konstanten Faktor ergeben. Diese Komplexität macht die meisten Probleme. Wenn die Ausführung eines Algorithmus' zeitlich das Vielfache des Alters des Universums erfordert, ist er hoffnungslos zeitaufwendig zu nennen.

8. Noch viel schneller als $O(2^n)$ wachsen Funktionen mit $O(n!)$.

9. Und nochmals schneller wachsen Funktionen mit $O(n^n)$.

10. Es gibt noch viel schlimmere Funktionen!

Zur Demonstration der Unterschiede der verschiedenen Zeitkomplexitäten sei die folgende Tabelle angeführt. Es sei hierzu angenommen, dass ein Computer eine Million einfache Anweisungen pro Sekunde ausführen kann. In der ersten Zeile ist die Eingabegröße (Anzahl der zu verarbeitenden Daten) angegeben. Die erste Spalte stellt die Zeitkomplexität dar:

	10	20	50	100	200
N^2	$\frac{1}{10000}$ s	$\frac{1}{2500}$ s	$\frac{1}{400}$ s	$\frac{1}{100}$ s	$\frac{1}{25}$ s
N^5	$\frac{1}{10}$ s	3,2 Min.	5,2 Min.	2,8 h	3,7 Tage
2^N	$\frac{1}{1000}$ s	1 s	35,7 Jahre	über 400 Billionen Jahrhunderte	45stellige Zahl an Jahrhunderten
N^N	2,8 h	3,3 Billionen Jahre	70stellige Zahl an Jahrhunderten	185stellige Zahl an Jahrhunderten	445stellige Zahl an Jahrhunderten

Zum Vergleich: Der Urknall war vor ca. 12-15 Milliarden Jahren.
Am wichtigsten ist wohl dabei die Unterscheidung zwischen *polynomialem* und *exponentiellem* Verhalten. Hat ein Algorithmus einen zeitlichen Aufwand $A(n)$ mit einem Polynom $A(n) = a_k n^k +$

$a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$ mit $a_k \neq 0$, dann hat der zugehörige Algorithmus die Zeitkomplexität $O(n^k)$, da die höchste Potenz für große n dominant ist.
In der Praxis hat sich gezeigt, dass die meisten polynomialen Probleme Algorithmen sind, die eine Zeitkomlexität von höchstens $O(n^3)$ haben. Übereinstimmend ist man der Meinung, dass höchstens polynomiale Algorithmen praktische Bedeutung haben. Algorithmen mit exponentiellem Rechenzeitaufwand sind für praktische Anwendungen nur bedingt geeignet.
Man könnte nun einwenden, dass zukünftige Computer sehr viel schneller sein werden. Das ist richtig. Aber was ändert sich dadurch an der Tabelle? Die Annahme war, dass eine Million Anweisungen pro Sekunde verarbeitet werden. Nehmen wir nun einen Computer an, der 10 000 mal schneller ist als heutige Computer. Die Entwicklung dazu wird doch noch einige Zeit dauern und viele Informatiker werden sich über diesen Fortschritt freuen. Die Einträge in der Tabelle werden sich allerdings nicht deutlich ändern. Der Eintrag „eine 185stellige Zahl an Jahrhunderten“ muss dann ersetzt werden durch die Angabe „eine 180stellige Zahl an Jahrhunderten“. Toller Fortschritt! Was könnte dann ein neuer Computer in der gleichen Zeit erledigen wie ein alter? Mit dem neuen Computer könnten wir gerade mal 102 statt 100 Eingabedaten in der gleichen Zeit verarbeiten. Mehr ist leider nicht drin!
Das zeitliche Verhalten der Algorithmen legt es nahe, die Algorithmen als *durchführbar (tractable)* oder *undurchführbar (intractable)* zu klassifizieren.
Ein Algorithmus gilt als

- durchführbar, wenn er ein polynomiales Verhalten wie $O(n^k)$ besitzt oder sich noch besser wie $O(\log n \cdot n)$ verhält.
- undurchführbar, wenn er sich wie $O(2^n)$, $O(n!)$, $O(n^n)$ oder noch schlimmer verhält.

Beispiele für undurchführbare Algorithmen:

- *Das Rundreiseproblem oder Problem des Handlungsreisenden (travelling salesman problem)*. Ein Vertreter muss n Städte besuchen, deren Entfernungen voneinander bekannt sind. Er besucht jede nur einmal und kehrt von der letzten zu seinem Ausgangspunkt zurück. In welcher Reihenfolge muss er sie besuchen, damit der gesamte von ihm zurückgelegte Weg ein Minimum ist[1]?

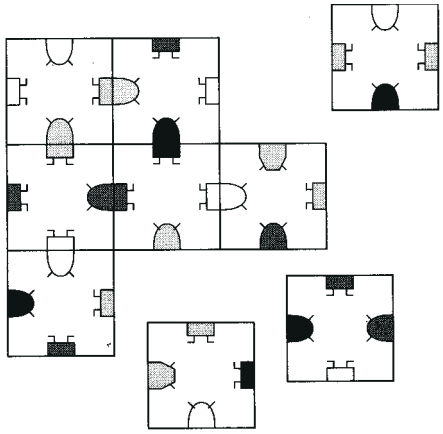
  Wenn ein Algorithmus sämtliche möglichen Routen durchläuft, kommt er auf eine Zeitkomplexität $O(n!)$. Bei 150 bis 200 Städten besteht dann keine Hoffnung mehr, eine Lösung zu bekommen.

- *Das Rucksackproblem (knapsack problem)*. Ein Rucksack soll mit einer Auswahl aus n gegebenen Gegenständen von verschiedenem Wert und verschiedenem Gewicht gefüllt werden. Wie muss die Auswahl geschehen, damit der Wert der Gegenstände im Rucksack ein Maximum ist, wenn dabei ein gegebenes Maximalgewicht des gefüllten Rucksacks nicht überschritten werden darf?

- *Das Stundenplanproblem* ist ein Beispiel für ein *scheduling*-Problem, von denen es viele ähnliche gibt. Die Zeitkomplexität ist exponentiell.

[1] Ein sehr schönes Buch dazu ist: Gritzmann/Brandenberg, Das Geheimnis des kürzesten Weges - Ein mathematisches Abenteuer, Springer-Verlag 2002, ISBN 3-540-42028-2

- *Das Affenpuzzle.* Es besteht aus neun quadratischen Karten, von denen jede der vier Seiten die obere oder die untere Hälfte eines farbigen Affen zeigt. Ziel des Spiels ist es, mit den Karten ein 3x3-Quadrat zu legen, bei dem an jeder Schnittstelle ein vollständiger und einfarbiger Affe entsteht.



  Das allgemeinere Problem wäre ein Spiel mit N Karten, wobei N eine Quadratzahl ist. Ein Algorithmus könnte sich nun durch alle möglichen Anordnungen durcharbeiten. Ist eine Anordnung zulässig, hält er mit der Ausgabe „ja“ an. Sind alle möglichen Anordnungen durchprobiert und keine als zulässig erkannt worden, hält er mit der Ausgabe „nein“ an.

  Was passiert nun z.B. bei einer 5x5-Anordnung mit 25 Karten? Für die erste Karte in der linken unteren Ecke gibt es 25 Möglichkeiten, eine Karte auszuwählen und 4 Möglichkeiten der Lage der Karte, was insgesamt 100 Möglichkeiten für den ersten Zug ergibt. Für die zweite Karte auf der nächsten Position gibt es 24 Möglichkeiten und wieder 4 Möglichkeiten der Lage, also 96 Möglichkeiten usw. Die Gesamtzahl der Anordnungen ist also

$$(25 \cdot 4) \cdot (24 \cdot 4) \cdot (23 \cdot 4) \cdot \ldots \cdot (3 \cdot 4) \cdot (2 \cdot 4) \cdot (1 \cdot 4),$$

  was man als $25! \cdot 4^{25}$ schreiben kann. Nehmen wir an, dass ein Computer in einer Sekunde 1 Million Anordnungen testen kann. Dann benötigt er im ungünstigsten Fall

  533.000.000.000.000.000.000.000.000 Jahre.

  Seit dem Urknall sind erst etwa 12-15 Milliarden Jahre vergangen. Mit einigen Tricks kann man den Algorithmus verbessern. Aber das hilft nicht viel. Eine Zeitkomplexität $O(N! \cdot 4^N)$ ist sozusagen doppelt bösartig.

- *Minesweeper* ist ein Spiel, das Windows-PCs beigelegt ist. Im Jahre 2000 wurde von RICHARD KAYE bewiesen, dass das Problem, für eine vorgelegte Minesweeper-Konstellation zu entscheiden, ob sie logisch konsistent ist, ein NP-Problem (s.u.) ist.

- *Das SAT-Problem oder satisfiability problem* war eines der wichtigsten Probleme in diesem Zusammenhang. Gegeben ist ein großer komplizierter Schaltkreis. An den Eingängen des Schaltkreises kann jeweils der Wert `true` oder `false` anliegen. Der Schaltkreis besteht aus logischen Schaltelementen wie `AND`, `OR` und `NOT`. Den vielen Eingängen ist über weitere Elemente ein einziger Ausgang nachgeschaltet. Gibt es nun für einen gegebenen Schaltkreis

eine Kombination von Eingängen in der Form von `true` oder `false`, so dass am einzigen Ausgang ein `true` erscheint? Bei einfachen Schaltkreisen ist das einfach. Wir haben das im Unterricht mit Beispielen zu diesem Thema kennen gelernt. Bei komplizierten Schaltkreisen wird es schier unlösbar. Interessant dabei ist, dass sich Elemente eines Schaltkreises in Minesweeper- Konstellationen bzw. in ein Minesweeper-Konsistenzproblem transformieren lassen. Das SAT-Problem ist das berühmteste NP-Problem.

- *Viele weitere Beispiele* gibt es in den Gebieten Kombinatorik, Graphentheorie, Spieltheorie, Logik, Wirtschaftswissenschaften, Telekommunikation, Compileroptimierung, Bankenwesen, Betriebsplanung, Stadtplanung, Logistik und beim Entwurf integrierter Schaltkreise.

NP-vollständige Probleme

Alle im letzten Abschnitt aufgeführten Probleme gehören zu einer Klasse von zur Zeit etwa 2000 sehr verschiedener algorithmischer Probleme, die alle genau die gleichen Phänomene aufweisen. Man nennt sie *NP-vollständige Probleme.*

- Sie sind entscheidbar.
- Sie besitzen Lösungen in exponentieller Zeit.
- Für kein Problem wurde je ein Algorithmus mit Polynomialzeit gefunden.
- Niemand konnte bisher beweisen, ob sie exponentielle Zeit benötigen müssen.
- In gewissem Sinn sind alle diese Probleme miteinander verwandt. Sollte jemals für ein einziges Problem ein Algorithmus mit Polynomialzeit gefunden werden, dann ergäben sich sofort Polynomialzeit-Algorithmen für **alle** anderen Probleme.

  Sollte jemals für eines der Probleme gezeigt werden, dass es keinen polynomialen Algorithmus geben kann, dann ist damit automatisch die Undurchführbarkeit für **alle** anderen Probleme bewiesen.

  Diese enge Verwandtschaft aller NP-vollständigen Probleme ist bewiesen worden (STEPHEN COOK, 1971)!
- Das Finden einer Lösung ist bei allen Problemen schwierig. Hat man aber einen Lösungskandidaten vorliegen, so kann mit nur polynomialem Zeitaufwand überprüft werden, ob die Lösung korrekt ist.
- Mit „Zauberei“ kann ein besserer Algorithmus gefunden werden. Was soll das? Nehmen wir an, dass wir beim Affenpuzzle alle Möglichkeiten durchprobieren. An Stellen des Programmes, wo es mehrere Möglichkeiten des Weitergehens gibt (z.B. beim Anlegen einer weiteren Karte), werfen wir eine „magische Münze“ einmal oder je nach Bedarf mehrmals und folgen ihrem Ergebnis. Die Münze fällt aber nicht zufällig, sondern besitzt magische Kenntnisse über den besten Weg. Der Fachausdruck für diese Art Magie heißt *Nicht-Determinismus*.

  Die NP-vollständigen Probleme zeichnen sich dadurch aus, dass es mit dieser Zauberei einen polynomialen Algorithmus gibt. Dies erklärt auch den Ausdruck NP: **n**icht-deterministisch **p**olynomial. Das Wort „vollständig“ steht für die vorher angesprochene enge Verwandtschaft.

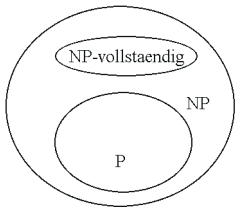
- Man beweist die NP-Vollständigkeit eines Problems dadurch, dass man es durch geeignete Tranformationen auf ein anderes Problem zurückführt, von dem schon bewiesen wurde, dass es NP-vollständig ist.

Das große Geheimnis: Gilt P=NP?

In der Informatik wurden für die verschiedenen Problemklassen Namen eingeführt:

- **P** steht für die Algorithmen mit polynomialer Zeitkomplexität, also die harmlosen.
- **NP** steht für die Algorithmen mit polynomialer Zeitkomplexität, wenn man Zauberei einsetzt.
- **NP-vollständig** steht für die „härtesten“ Probleme in NP. Falls eines der Probleme in P liegen sollte, dann auch alle anderen und auch alle anderen aus NP.

Offensichtlich gilt $P \subseteq NP$, denn jeder deterministische Algorithmus (ohne Zauberei) ist ein Spezialfall eines nichtdeterministischen. Ein offenes Problem der Informatik ist, ob die Inklusion echt ist.



Würde man irgendwann einmal für ein einziges NP-vollständiges Problem einen polynomialen Algorithmus finden, fiele P mit NP zusammen. Die Frage, ob P=NP ist, wird heute von vielen Informatikern mit an Sicherheit grenzender Wahrscheinlichkeit verneint.
Das Clay Mathematics Institute, eine gemeinnützige Bildungseinrichtung in Cambridge (Massachusetts), hat Preise von jeweils einer Million Dollar für die Lösung von sieben prominenten mathematischen Problemen ausgesetzt. Eines davon ist die Frage, ob P=NP ist.

Praktische Konsequenzen

Es gibt viele Schulen, die einen Stundenplan haben, dessen Berechnung nicht Quadrilliarden von Jahren benötigt hat. Es sind eben nicht perfekte Stundenpläne, die berechnet werden, sondern nur Näherungslösungen. Bestimmte Vorgaben konnten vielleicht nicht berücksichtigt werden, aber das ist immer noch besser als das totale Chaos im Schulgebäude.
Für das Problem des Handlungsreisenden gibt es eine Näherungslösung, die mit $O(n^3)$ läuft und garantiert, dass der gefundene Weg nicht mehr als 50% länger als der unbekannte optimale Weg ist.

Ausblick

Die bisherigen Ergebnisse sind durchaus sehr ernüchternd. Computer können die meisten Probleme überhaupt nicht lösen und für viele interessante Probleme benötigen sie fast unendlich viel Zeit.
Als ein kleines Licht am Ende des Tunnels könnten sich zwei Entwicklungen erweisen, die forschungsmäßig noch in den Kinderschuhen stecken: Quantencomputer und Molekularcomputer. Auf eine weitergehende Erklärung sei hier verzichtet. Mit Sicherheit kann man sagen, dass diese Computer die bisher unlösbaren Probleme auch nicht lösen. Eine gewisse Chance gibt es aber, dass die „harten“ NP-vollständigen Probleme durch sie unter einem neuen Licht erscheinen könnten.