Grenzen der Berechenbarkeit

Analyse der Laufzeiten von Algorithmen

Eine sehr präzise oder sogar exakte Analyse der Laufzeit wird im Allgemeinen nicht durchgeführt. Es wäre bei komplexeren Algorithmen mit unterschiedlichen Verläufen durch ***Fallunterscheidungen oder bei Nutzung von Zufallszahlen*** mathematisch schwierig zu bestimmen. Zudem müsste man eine isolierte Betrachtung ohne Zwischenfunken des Betriebssystems oder von anderen Programmen ermöglichen. Auch das ist schwer realisierbar. Dennoch würde man gerne ***Algorithmen klassifizieren***, auch um eine Aussage über deren ***Effizienz*** bezüglich ihrer Berechnung zu treffen. Daher ist es notwendig einige Abstraktionsschritte vorzunehmen:

- Die Arten der Operationen (Zuweisungen, Vergleiche, Arrayzugriffe,…) wird ***nicht unterschieden***, auch wenn die auf Maschinenebene unterschiedlich aufwendig wären
- Die Menge aller Eingaben muss ***allgemein betrachtet*** werden. Daher wird die Größe der Eingabe mit der Variablen ***n*** versehen (z. B. n-elementiges Array).
- Betrachtung der Größenordnung der Laufzeit mittels eine Laufzeitfunktion ***T(n)*** durch ***Weglassen additiver und multiplikativer Konstanten***.

Das O-Kalkül

Der exakte Aufwand eines Algorithmus (z. B. als Anzahl Rechenschritte bis zur Terminierung oder als verbrauchte Zeit) kann in der Regel nur schwer als Funktion von Umfang und Eingabewerten allgemein angegeben werden. ***Wichtiger ist die ungefähre Größenordnung, mit der der Aufwand in Abhängigkeit vom Umfang der Eingabe wächst.*** Das ***O-Kalkül*** ist ein einfaches Mittel, um Algorithmen in bestimmte ***Laufzeitklassen*** einzuordnen.

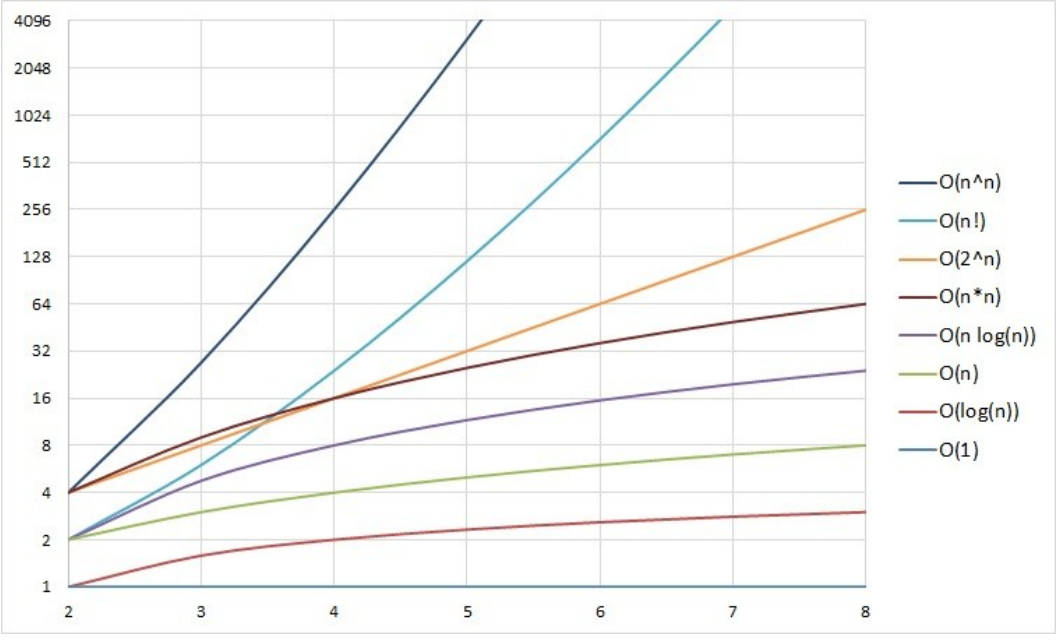
Man trifft durch die drei oben beschriebenen Abstraktionsschritte eine Abschätzung des Programmaufwands bzw. einer Funktion im Bezug auf den Eingabeumfang durch eine asymptotische „Vergleichsfunktion“.

Das heißt, dass der Rechenaufwand des zu untersuchenden Programms bzw. der Funktion bei Vergrößerung der Eingabemenge ***nicht wesentlich schneller wächst*** als meine Vergleichsfunktion. Somit ist dann das zu untersuchende Programm ***in derselben Laufzeitklasse*** wie meine Vergleichsfunktion.

Mathematisch würde man das folgendermaßen beschreiben:

$f(n)$ wächst nicht wesentlich schneller als $g(n)$ (mathematisch: $f(n) \in O(g(n))$), falls es eine Konstante c gibt, so dass für große n immer $f(n) \leqslant c * g(n)$ gilt.

Laufzeitklasse	Sprechweise	Typische Algorithmen dieser Laufzeitklasse
O(1)	konstant	
$O(\log_2 n)$	logarithmisch	
$O(\sqrt{n})$		
O(n)	linear	
$O(n * \log_2 n)$		
...		
$O(n^2)$	quadratisch	
$O(n^k)$, k>1	polynomiell	
...		
$O(b^n)$, b>1	exponentiell	
O(n!)	faktoriell	
$O(n^n)$		



Das Halteproblem

Bei Algorithmen, die sehr lange zur Berechnung (> ein Menschenleben) brauchen, kann man sich die Fragen stellen, ob diese Algorithmen jemals terminieren (= sich beenden mit positiven oder negativem Ergebnis). Für viele Algorithmen kann man diese Frage leicht beantworten.

Alan Turing konnte allerdings beweisen, dass es keinen Algorithmus geben kann, der entscheiden/berechnen kann, ob ein gegebener Algorithmus hält/terminiert oder nur sehr lange/unendlich lange rechnet.

Das Halteproblem ist nicht berechenbar (und auch nicht entscheidbar).

(Für Profis: Berechenbarkeit wird über ein Berechnungsmodell bewiesen (WHILE-berechenbar, Turing-berechenbar, µ-rekursiv-berechenbar, intuitiv-berechenbar, ….). Entscheidbarkeit bedeutet, dass für ein Problem (z. B. Ist eine Zahl eine Primzahl? Oder gibt es ein Programm, dass entscheiden kann ob ein Algorithmus hält?) ein Algorithmus existiert, der für eine bestimmte Eingabe entscheiden: ja oder nein.)

Beweis über ein Gegenbeispiel:

Angenommen man hätte den Halteproblemalgorithmus H(Programm P, Eingabewerte E), der für einen anderen gegebenen Algorithmus/Programm mit bestimmten Eingabewerten entscheiden kann, ob dieses terminiert oder nicht. Das heißt:

- ***H(P, E) = true, falls das Programm mit Eingabewerten terminiert***
- ***H(P, E) = false, falls das Programm mit Eingabewerten nicht terminiert***

→ Da dieser Algorithmus universell funktionieren muss, kann man jetzt dem Halteproblemalgorithmus ein gegebenes Programm sowohl als Programm als auch als Eingabe (z. B. binär kodiert) übergeben → H(P, P). So muss der Halteproblemalgorithmus auch in endlicher Zeit entscheiden können, ob P mit der Eingabe P hält oder nicht.

Man definiert sich ein weiteres Programm ***M(Programm P) = H(P, P) mit dem Programmcode m***. Das Programm M soll das Komplement (= Gegenteil) von H darstellen:

- M(P) hält, falls H(P, P) = false ist. Sprich:
- M(P) hält nicht (z. B. durch Endlosschleife mit while(true){ }), falls H(P, P) = true

→ spezielles Halteproblem: Was ist dann die Ausgabe von M(m) = H(m,m)?

- **M hält bei Eingabe m: H(m,m) = true → M(m) Endlosschleife → Widerspruch**
- **M hält bei Eingabe m nicht: H(m,m) = false → M(m) hält → Widerspruch**

→ Ein solches Programm kann es nicht geben!

Nochmal visualisiert: https://www.youtube.com/watch?v=hA_BIVzyh4Y