Asymptotische Komplexität

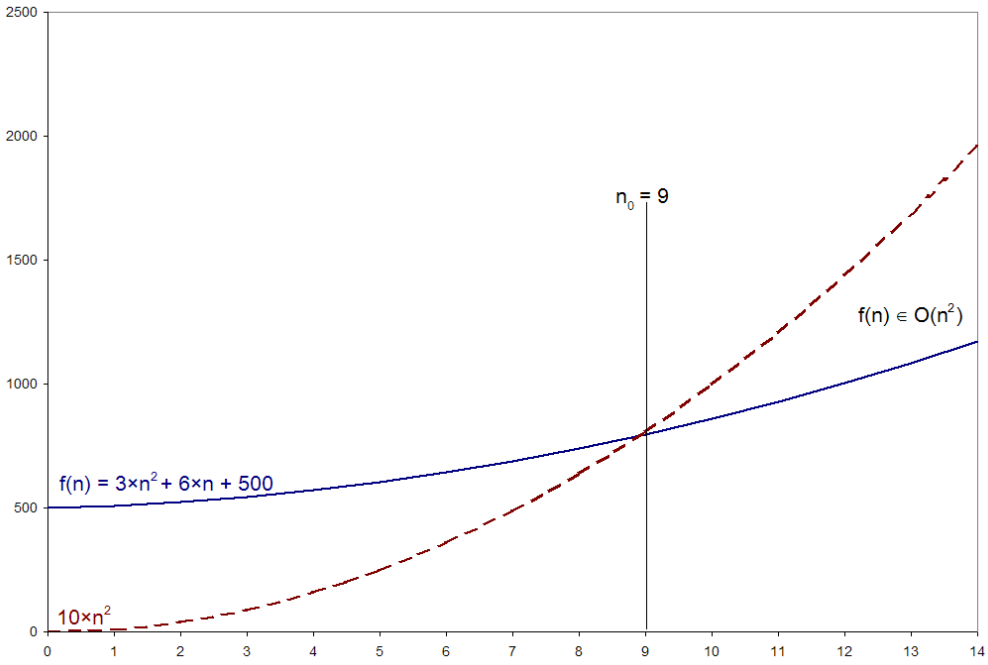
1. Definition

Die Beispiele des Arbeitsblatts *Instruktionen Zählen* zeigen, dass sich die Laufzeit von Algorithmen sehr unterschiedlich verändert, wenn die Grösse der zu bearbeitenden Probleme immer weiter zunimmt bzw. asymptotisch ins unendliche geht. Um diese deutlichen Unterschiede fassbar zu machen, möchten wir Algorithmen in *Komplexitätsklassen* einteilen, welche die Zunahme des Aufwands für immer grösser werdende Problemgrössen widerspiegeln.

Die Laufzeit von Programmen lässt sich grundsätzlich als Funktion der Problemgrösse ausdrücken: *T(n)*. Einen Weg, um von einem Programm zu so einer Funktion zu kommen, haben Sie gesehen: Man zählt, wie oft die am häufigsten ausgeführte Operation durchgeführt wird.

Um Funktionen zu klassifizieren, führen wir folgende Notation ein (*O-Notation*):

$$f(n) \in O(g(n)) \stackrel{\text{def}}{=} \left(\exists\, c, n_0 : c > 0, n_0 > 0 : \left(\forall\, n : n \geq n_0 : f(n) \leq c \cdot g(n)\right)\right)$$



$$f(n) \in O(n^2) \; mit \; c = 10, n_0 = 9$$

Um zu entscheiden, ob eine Funktion in einer bestimmten Funktionsklasse liegt, muss man entweder passende Werte für *c* und n_0 finden, oder beweisen, dass es keine solchen gibt.

Beispiel:

a) Gilt $2n + 5 \in O(n^2)$?

c = n_0 =

z.B. c = 10 und n0=10

somit gilt:

2n+4 <= c0*n^2

mit den Zahlen eingesetzt:
2*10+4 <= 10*10^2
24 <= 1000

b) Gilt $2n + 5 \in O(n)$?

c = n_0 =

Obige Beispiele erlauben 2 Beobachtungen:

- Wenn überhaupt, gibt es unendlich viele Möglichkeiten, c oder n0 zu wählen
- O(n) ist eine Teilmenge von O(n^2): Interessant ist also die kleinste passende Komplexitätsmenge

Grundlegendes Vorgehen:
1. Häufigste Anweisung suchen
2. Funktion f(n) beschreibt Anzahl Ausführungen abhängig von Problemgrösse n
3. Komplexitätsklasse von f(n) bestimmen
4. Komplexitätsklasse liefert Erkenntnisse

2. Rechenregeln für die O-Notation

Das direkte Hantieren mit der Definition ist im Allgemeinen eher mühsam. Deswegen erleichtert man sich mit einigen Rechenregeln die Arbeit. Besonders nützlich sind die folgenden 4, die sich alle direkt aus der Definition der *O-Notation* ableiten lassen:

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Leftrightarrow f(n) \in O\big(g(n)\big)$
2. $k \cdot f(n) \in O\big(f(n)\big)$
3. $f(n) \in O\big(g(n)\big) \Rightarrow f(n) + g(n) \in O\big(g(n)\big)$
4. $f(n) \cdot g(n) \in O\big(f(n) \cdot g(n)\big)$

Beispiele:

Gilt $3^n \in O(2^n)$?

Exponentialfunktionen mit grösserer Basis wachsen schneller!

Gilt $2^n \in O(n^k)$?

Regel 1 lässt sich evtl. mit dem Satz von d'Hospital[1] kombinieren.

Exponentialfunktionen wachsen schneller als jedes Polynom

Gilt $\log_{10} n \in O(\log_2 n)$? [2]

Die Basis von Logarithmen ist bei der O-Notation egal und wird weggelassen

Zu welcher Klasse gehört $4n^3 + 7n^2 + 2n + 1$?

[1] $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$, falls $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$ oder $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = 0$

[2] Diese Fragestellung lässt sich gut mithilfe des *Basisaustauschsatzes* für Logarithmen beantworten: $log_a(n) = \frac{log_b(n)}{log_b(a)}$

3. Wichtige Klassen

Klasse	Bezeichnung	$\frac{f(2n)}{f(n)} \approx$	Zuwachs
$\mathcal{O}(1)$	konstant	1	kein Zuwachs
$\mathcal{O}(\log n)$	logarithmisch	$1 + \frac{\log 2}{\log n}$	Zuwachs nimmt mit grösseren *n* ab
$\mathcal{O}(n)$	linear	2	Zuwachs mit konstantem Faktor (unabhängig von *n*)
$\mathcal{O}(n \log n)$		$2 + \frac{2 \cdot \log 2}{\log n}$	
$\mathcal{O}(n^2)$	quadratisch	4	
$\mathcal{O}(n^3)$	kubisch	8	
$\mathcal{O}(2^n)$	exponentiell	2^n	Zuwachs nimmt mit grösserem *n* zu
$\mathcal{O}(3^n)$		3^n	
$\mathcal{O}(n!)$	Fakultät	$\prod_{i=n+1}^{2n} i$	

4. Asymptotische Komplexität einiger Algorithmen

4.1. Multiplikation quadratischer Matrizen

```
mul_mat = []
for i in range(len(m1)):
   row = []
   for j in range(len(m2[0])):
      s = 0
      for k in range(len(m2)):
         s += m1[i][k] * m2[k][j]
      row.append(s)
   mul_mat.append(row)
```

4.2. Sequenzielle Suche

```
i = 0
while i < len(data) and data[i] != key:
   i = i + 1
```

4.3. Maximum Sub Sequence by Divide and Conquer

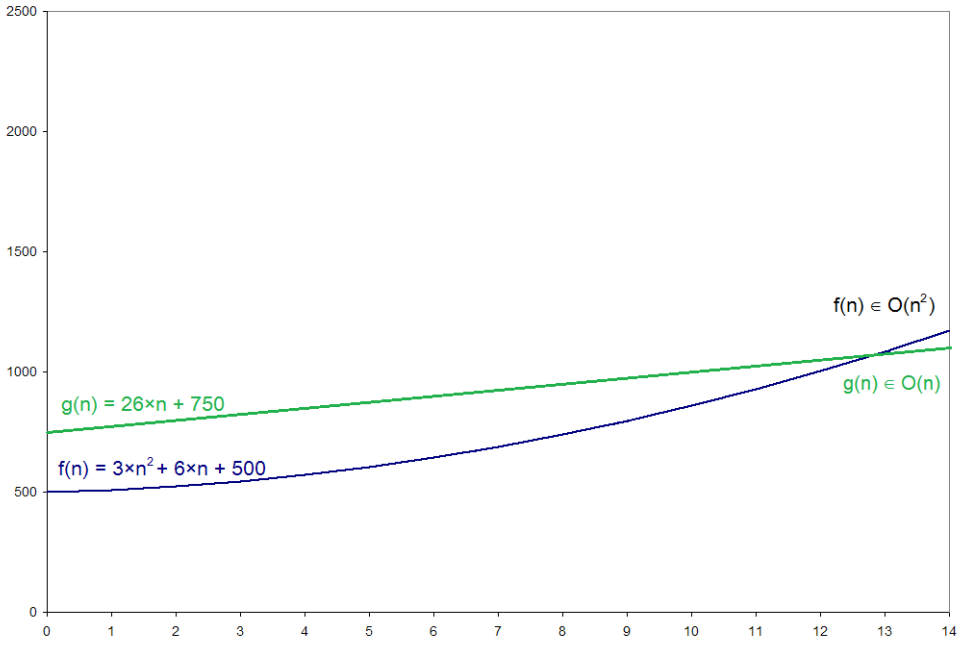
```
def max_sub_seq_divide_and_conquer(data):
   def max_in_range(r):
      max_sum, s = 0, 0
      for i in r:
         s = s + data[i]
         if s > max_sum:
            max_sum = s
      return max_sum

   def max_sub_seq_part(beg, end):
      if beg == end:
         return max(0, data[beg])
      else:
         m = (beg + end) // 2
         return max(max_sub_seq_part(beg, m), max_sub_seq_part(m + 1, end),
               max_in_range(range(m, beg-1, -1)) + max_in_range(range(m+1, end+1)))

   return max_sub_seq_part(0, len(data) - 1) if len(data) > 0 else 0
```

5. Bemerkung: Asymptotische Komplexität sagt nur etwas für grosse Problemgrössen

Das Mass der Asymptotischen Komplexität mit seiner *O*-Notation wurde bewusst so definiert, dass es für Problemgrössen, die gegen unendlich wachsen etwas aussagt. Das passt insofern zur Praxis, als dass in der Tat Algorithmen auf immer grössere Probleme angewendet werden und ein Mass für den Zuwachs der Rechenzeit (oder des Bedarfs an anderen Ressourcen wie Speicher) dann wichtig ist.

Ausserdem ist oft der Grenzwert n_0 im Sinne der Definition der *O*-Notation genügend klein, um von realen Problemgrössen bereits überschritten zu werden.

Andererseits erklärt gerade dieser Grenzwert, warum für genügend kleine Probleme ein asymptotisch effizienter Algorithmus langsamer sein kann als ein asymptotisch ineffizienter. Rechnen Sie mit dieser Möglichkeit!



6. Arten der Analyse

Fast alle bisher in diesem Kapitel betrachteten Algorithmen laufen für eine gegebene Problemgrösse immer gleich lang, unabhängig von den verwendeten Daten. Die einzige Ausnahme ist die Sequenzielle Suche, die je nach Fundort gleich am Anfang, irgendwann „mittendrin“ oder erst ganz am Ende abbricht.

In solchen Fällen muss man überlegen, was man genau wissen möchte. Benötigt man eine garantierte obere Grenze für die maximale Laufzeit, das durchschnittliche Verhalten, oder möchte man wissen, wann ein Ergebnis frühestens zur Verfügung stehen kann?

Je nach Art der Fragestellung unterscheidet man zwischen:

Worst Case Analyse	$T_{worst}(n)$	Aufwand im schlimmsten Fall, eine obere Schranke für den Aufwand
Average Case Analyse	$T_{avg}(n)$	Durchschnittlicher Aufwand bei häufiger Benutzung des Algorithmus Hängt von der Wahrscheinlichkeitsverteilung der Daten ab und ist meist nur mit grossem Aufwand genau zu berechnen
Best Case Analyse	$T_{best}(n)$	Aufwand im besten Fall, eine untere Schranke für den Aufwand Schneller geht es nicht

7. Ω- und Θ-Notation (Omega- und Theta-Notation)

Neben der *O-Notation*, die eine *obere Schranke* für eine Funktion angibt, gibt analog die *Ω-Notation* eine *untere Schranke* an: $f(n) \in \Omega(g(n)) \stackrel{\text{def}}{=} \left(\exists\, c, n_0 : c > 0, n_0 > 0 : \left(\forall\, n : n \geq n_0 : c \cdot g(n) \leq f(n)\right)\right)$.

Die *Θ-Notation* gibt eine obere und untere Schranke für das asymptotische Verhalten einer Funktion an. Es gilt: $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$ bzw. $f(n) \in \Theta(f(n)) \equiv f(n) \in O(f(n)) \wedge f(n) \in \Omega(f(n))$.

Ein Beispiel: Sei $f(n) = 2n^2 + 3n + 5$ dann gilt u.a.: $f(n) \in O(n^3)$, $f(n) \in \Omega(n)$ aber nur $f(n) \in \Theta(n^2)$.