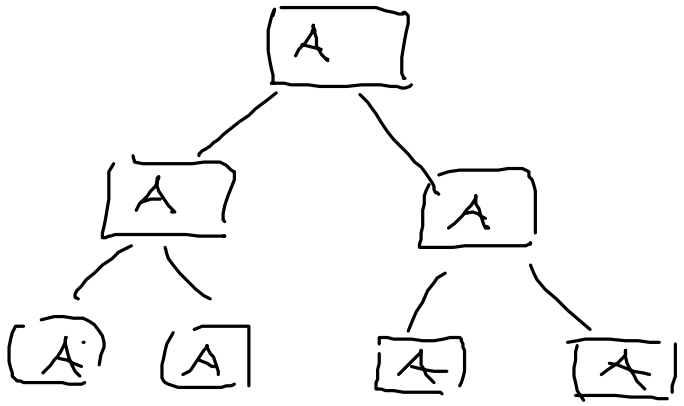
Kap 8. Rekursion

Rekursion:

Methoden können sich selbst aufrufen

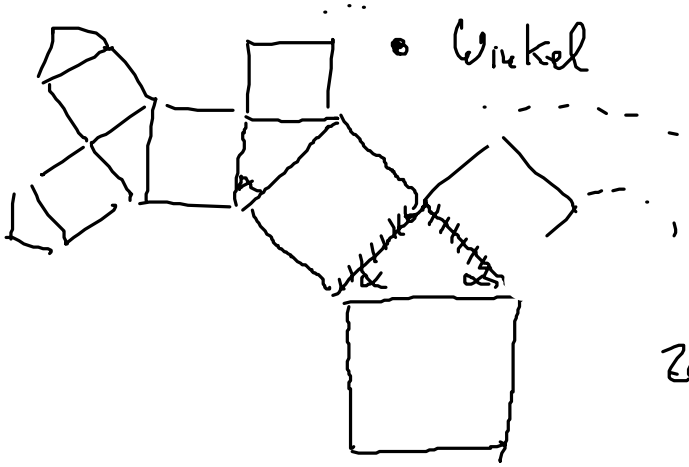


Rekursionstiefe
= Höhe (Rekursionsbaum)
+1
≙ Maß für Speicher

Knoten ≙ Maß für Laufzeit

Bsp: Pythagoras-Bäume

Methode: Input: • Strecke im $\mathbb{R}^2$ (Seitenlänge Quadrat)
• Winkel



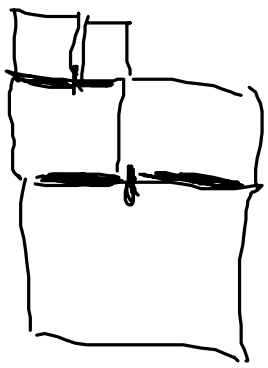
Zeichnung ≙ Rekursionsbaum umgedreht

Funktionsweise: Zeichnet Quadrat + Dreieck mit Winkel α

+ rekursive Aufrufe für die Seiten des Dreiecks, die nicht zum Quadrat gehören

nichts machen, falls Länge Strecke < 2mm ist

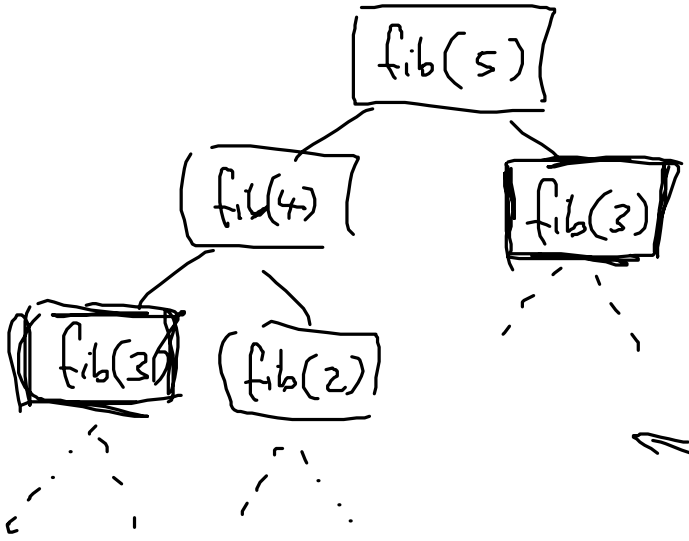
$\tan(\alpha) \approx 0$



Fibonacci Zahlen

$f_0 = 0 \quad f_1 = 1 \qquad f_{n+1} = f_n + f_{n-1}$

0 1 1 2 3 5 8 ...



Schlecht,
viele Berechnungen
werden mehrfach
gemacht

hier Iteration besser

0 1 1 2 3 5 8 13

x y z

z = x + y

x = y

y = z

Kap 9 Die Analyse von Algorithmen

Ziel Effizienzuntersuchung von Algorithmen

↑

unabhängig von Technologie

brauchen dafür ein "Rechnermodell"

RAM - Modell Random Access Machine

- random access auf elementare Datentypen zählt 1
  und Komponenten von strukturierten Datentypen
  mit random access (Arrays, Klassen)
- können beliebig lange Zahlen speichern
- Zahlen (je nach Anwendung)
  arithmetische Operationen, Vergleiche, Zuweisungen ...
  meist alle mit 1 gezählt

9.1. Analysearten

- A priori Analyse
  - worst case analysis
    obere Schranke für alle Inputs
    ^
    der Laufzeit (gemessen in typischen Operationen)

Bsp: Suchen eines Wertes in Array mit n Komp.

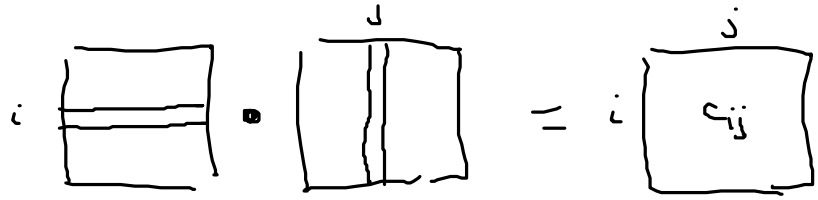
worst case: n Vergleiche

- average case analysis

  mittelt über alle möglichen Inputs

- untere Schranken für Laufzeit

Bsp: Matrizenmultiplikation von $n \times n$ Matrizen

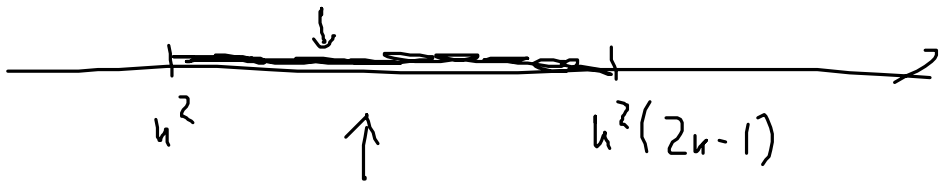


$$c_{ij} = \underbrace{\sum_{k=1}^{n} a_{ik} \cdot b_{kj}}_{\text{n Mult, n-1 Add.}}$$

n^2 Ergebnis-Elem.

$\Big\} \; n^2(2n-1)$

natürliche untere Schranke: n^2 (da n^2 Ergebniszahlen berechnet werden müssen)



Bereich für Laufzeit von Algos

- A posteriori Analyse

Laufzeitmessungen mit Implementation des Algo

abh. von Prog.-Sprache, Rechner, Compiler

A priori Analyse interessiert sich nicht um "Konstanten"

nur Größenordnungen interessieren

↓

wie messen wir Größenordnungen?

9.2 Die asymptotische Notation

Betrachten Funktionen $f : \mathbb{N} \to \mathbb{N}$

↑ ↑

da Laufzeitfkt von Algorithmen

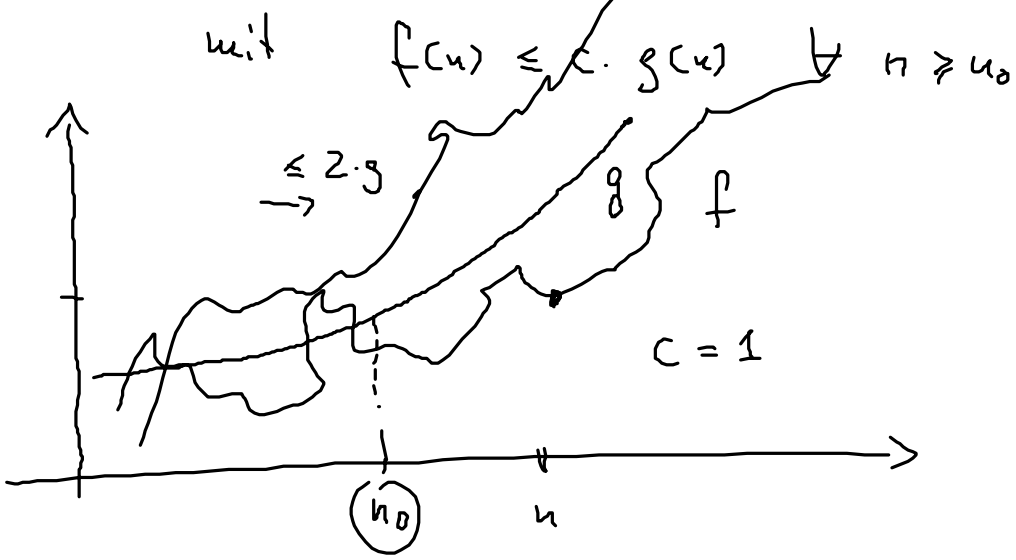
Sei $g : \mathbb{N} \to \mathbb{N}$

$$O(g) := \{ f : \mathbb{N} \to \mathbb{N} \mid \exists c \; \exists n_0 \in \mathbb{N} : \forall n \geq n_0 \; f(n) \leq c \cdot g(n) \}$$

↑

zu einem solchen f existiert Konstante c, Zahl n_0

(c, n_0 abh. von f)



Sprechweisen: f ist in $O(g)$ von g, f ist von der Größenordnung g

Schreibweisen: $f(n) \in O(g(n))$ $\quad f(n) = O(g(n))$

<u>SATZ</u>: Sei f ein Polynom, $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ ($+a_0$)
$a_m \neq 0$

[Bsp $f(n) = 3n^4 - 7n^3 + n - 17$]

Dann gilt: $f \in O(n^m)$

[Bsp $f \in O(n^4)$]

<u>Beweis</u>: Überprüfen Def von $O(g)$

$$f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$$

$$\leq |a_m| n^m + |a_{m-1}| n^{m-1} + \ldots + |a_1| n + |a_0|$$

$$= \underbrace{\left(|a_m| + \frac{|a_{m-1}|}{n} + \dots + \frac{|a_1|}{n^{m-1}} + \frac{|a_0|}{n^m} \right)} n^m$$

$n \geq n_0 := 1$

$$\leq C := |a_m| + |a_{m-1}| + \dots + |a_0|$$

$\Rightarrow f(n) \leq \underbrace{C \cdot n^m}_{=}$ für alle $n \geq \underbrace{n_0 = 1}_{=} \Rightarrow f \in O(n^m)$

$100\, n^2$ $\quad$ $\frac{1}{1000}\, n^3$

$\downarrow$ $\quad$ $\downarrow$

$O(n^2)$ $\quad$ $O(n^3)$

für große n sind kleinere Größenordnungen zu bevorzugen

Wie kann man Größenordnungen unterscheiden?

Mit O-Notation

Sei $g : \mathbb{N} \to \mathbb{N}$

$$O(g) = \{ f : \mathbb{N} \to \mathbb{N} \mid \forall c \ \exists n_0 \in \mathbb{N} : f(n) \leq c \cdot g(n) \quad \forall n \geq n_0 \}$$

$\equiv$

Sprechweise f ist klein oh von g

f ist von kleinerer Größenordnung als g

Schreibweisen $f(n) \in O(g(n))$ $f(n) = O(g(n))$ ...

Gängige Größenordnungen:

$$O(1) < O(\log n) < O(n) < O(n \cdot \log n) < O(n^2) < O(n^3).$$

$$< O(n^k) \text{ für festes } k$$

$$< O(n^{\log n}) < O(2^n)$$

polynomiale Größenordnungen

"$<$" $\hat{=}$ $f \in O(g)$

Algo von dieser Größenordnung heißt "von polynomialer Laufzeit" oder "effizient"

Keine polynomiale Laufzeit

$O(2^n)$ $\hat{=}$ exponentielle Laufzeit

Wie feststellen ob $f \in O(g)$?
Nützliches Kriterium der Analysis

SATZ: Seien $f, g : \mathbb{N} \to \mathbb{N}$. Dann gilt:

$$f \in O(g) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Beweis: "$\Rightarrow$" Sei $\varepsilon > 0$ beliebig, fest

Def von $O(g)$ $\Rightarrow$ zu ε $\exists$ n_0 mit $f(n) \leq \varepsilon \cdot g(n)$ $\forall n \geq n_0$

$\Rightarrow$ $\frac{f(n)}{g(n)} \leq \varepsilon$ $\forall n \geq n_0$

$\Rightarrow$ $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

"$\Leftarrow$" Sei $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$ $\Rightarrow$ zu $c > 0$ $\exists$ n_0 mit

$\frac{f(n)}{g(n)} \leq c$ $\forall n \geq n_0$

$\Rightarrow$ $f(n) \leq c \cdot g(n)$ $\forall n \geq n_0$ $\Rightarrow$ $f \in O(g)$ $\square$

Beispiel: Unterschied polynomielle Laufzeit vs. exponentielle Laufzeit

haben Problem mit Inputgröße n, Laufzeitfkt $f(n)$

heutiger Rechner braucht 1 Stunde für Problemgröße n_0

$f(n_0) = 1$ Std

Besserer Rechner 100 mal so schnell

$$f(n_{neu}) = 1\ \text{Std} = 100 \cdot f(n_0)$$

↑ gesucht

$$\Rightarrow f(n_{neu}) = 100 \cdot f(n_0)$$

Fall 1: f ist polynomial, etwa $f(n) = n^3$

$$n_{neu}^3 = 100 \cdot n_0^3 \Rightarrow n_{neu} = \sqrt[3]{100}\, n_0$$

$\sqrt[3]{100}$: 4, ...

Faktor

Fall 2: f ist exponentiell, etwa $f(n) = 2^n$

$$2^{n_{neu}} = 100 \cdot 2^{n_0}$$

nur additiv

$$n_{neu} = \log(100) + n_0$$

$\log(100)$: zwischen 6 und 7