Kapitel 2: Effizienz und Komplexität

Effizienz und Laufzeiten
O-Notation
Rekursion

DBS
Prof. Dr. Thomas Seidl

Effektivität und Effizienz von Algorithmen

- Effektivität
  - Ein Algorithmus liefert die gewünschten Ergebnisse
  - Beispiel Sieb des Erathostenes
    - Korrektheit: alle ausgegebenen Zahlen sind Primzahlen
    - Vollständigkeit: in der Ausgabe fehlen keine Primzahlen
  - „Die richtigen Dinge tun", das **Ziel** wird erreicht,

- Effizienz
  - Wirtschaftlichkeit im Ressourcenverbrauch
    - Beispiele: Laufzeit, Speicher, Kommunikation, Energie, etc.
  - „Die Dinge richtig tun", keine Ressourcenverschwendung, nicht zu viel **Aufwand** betreiben.

Effizienz von Algorithmen

- Effizienzfaktoren
  - Rechenzeit (Anzahl der Einzelschritte)
  - Speicherplatzbedarf
  - Zugriffe auf Sekundärspeicher (z.B. Festplatte, I/O)
  - Kommunikationsaufwand (z.B. Netzwerk, I/O)
  - Energieverbrauch; Kühlung für Abwärme

- konkrete Laufzeit eines Algorithmus hängt von vielen Faktoren ab
  - Takt der CPU
  - Länge der Eingabe
  - Implementierung der Basisoperationen

- Nutze abstrakte Rechnermodelle für Vergleiche

Komplexität

- Abhängig von Eingabedaten
- abstrakte Rechenzeit $T(n)$ abhängig von n:

Problem	$\boldsymbol{T(n)}$
Suchen in n-elementiger Menge	# Vergleiche, # zu durchlaufender Knoten
Sortieren einer n-elementigen Liste	# Vertauschungen/Vergleiche
Auswertung einer rekursiven Funktion $f(n)$	# Funktionsaufrufe
Finden aller Primzahlen bis n	# Rechenoperationen
Matrixmultiplikation $\mathbb{R}^{m\times k} \cdot \mathbb{R}^{k\times n}$	# Skalare Multiplikationen

- Üblicherweise asymptotische Betrachtung

Beispiel: Einfügen eines Stackelements

- Neue Liste initialisieren
- Speicher für Pointer „stack“ allokieren
- Speicher für Liste allokieren
- Liste initialisieren
  - value = null, next = null
- Listenpointer an „stack“ übergeben
- value-Attribut setzen
- next-Attribut setzen
- first-Attribut setzen

```
void push (Object v) {
  stack = new List();
  stack.value = v;
  stack.next = first;
  first = stack;
}
```

- Alle Operationen benötigen konstant viel Aufwand, unabhängig vom einzufügenden Wert
- Die Methode benötigt damit immer konstant viel Aufwand

Beispiel: SummeBis(n)

- Variable summe deklarieren und initialisieren
- 1 zur Summe addieren
- 2 zur Summe addieren
- 3 zur Summe addieren
- …
- n zur Summe addieren
- Ausgabe der Summe

```
int summeBis(int n) {
  summe = 0;
  for(int i = 0; i <= n; i++)
    summe += i;
  return summe;
}
```

- Alle Operationen benötigen konstant viel Aufwand
- Die Anzahl der Operationen wächst mit der Eingabe n
- Für linear wachsende Eingabe wächst der Aufwand ebenfalls linear

Beispiel: Matrixoperationen

```
Addition zweier Matrizen
Eingabe: A ∈ n × n, B ∈ n × n

Algorithmus:
Erzeuge Matrix C ∈ n × n
Für alle c_ij:
   c_ij = a_ij + b_ij
Gib C aus
```

Addition zweier Matrizen
Eingabe: $A \in n \times n$, $B \in n \times n$

Algorithmus:
Erzeuge Matrix $C \in n \times n$
Für alle c_{ij}:
$c_{ij} = a_{ij} + b_{ij}$
Gib C aus

Multiplikation zweier Matrizen
Eingabe: $A \in n \times n$, $B \in n \times n$

Algorithmus:
Erzeuge Matrix $C \in n \times n$
Für alle c_{ij}:
$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
Gib C aus

- Addition: Für n^2 viele Zellen wird eine Addition ausgeführt
- Multiplikation: Für n^2 viele Zellen werden n Multiplikationen und n Additionen ausgeführt: $(n^2 \cdot 2n)$
  (nicht-quadratische Matrizen komplizierter, aber analog)
- Der Aufwand für Matrixadditionen wächst quadratisch
- Der Aufwand für Matrixmultiplikationen wächst
  (etwas mehr als) kubisch

Asymptotische Komplexitätsklassen

- Wie verhalten sich Laufzeiten $T(n)$ für sehr große Eingaben $n \in \mathbb{N}$?
  - Maß für Komplexität soll unabhängig von konstanten Faktoren und Summanden sein.
  - Rechnergeschwindigkeit, Aufwände für Initialisierung etc. sollen ausgeklammert werden.

Formal: O-Notation

$$O(f) = \{g: \mathbb{R} \to \mathbb{R} \mid \exists c > 0\colon \exists x_0 > 0\colon \forall x \geq x_0\colon |g(x)| \leq c \cdot |f(x)|\}$$

In der Informatik liegen meist diskrete Eingaben aus $\mathbb{N}$ vor:

$$O(f) = \{g: \mathbb{N} \to \mathbb{R} \mid \exists c > 0\colon \exists n_0 > 0\colon \forall n \geq n_0\colon |g(n)| \leq c \cdot |f(n)|\}$$

Sprechweise: f ist obere Schranke von g

g wächst höchstens so schnell wie $O(f)$

Beispiel: Komplexitätsklasse für SummeBis(n)

```
int summeBis(int n) {
  summe = 0;                      // 1 Operation
  for(int i = 0; i < n; ++i) // 1+2n Operationen
    summe += i;                   // je 1 Operation
  return summe;                   // 1 Operation
}
```

- Damit ergeben sich $2 + n \cdot 3 + 1 = 3\text{n} + 3$ viele Operationen
- Erinnerung:
  $O(f) = \{g: \mathbb{N} \to \mathbb{R} \mid \exists c > 0{:}\ \exists n_0 > 0{:}\ \forall n \geq n_0{:}\ |g(n)| \leq c \cdot |f(n)|\}$
- $g(n) = \ n + 2$
  $3n + 3 \leq 4n$ für $c = 4$, $n_0 = 3$ und $f(n) = n \ \Rightarrow g \in O(n)$
- Es gibt unendlich viele Alternativen, c zu wählen, z. B. $c = 1001$:
  $n + 2 \leq 1001 \cdot n$ für $c = 1001$, $n_0 \geq 1$ und $f(n) = n \ \Rightarrow g \in O(n)$

O-Notation Rechenregeln: Konstanten

- Wenn $g(x) = a \in \mathbb{R}$ eine konstante Funktion ist, dann gilt

$$g \in O(1)$$

- Beweis

  Wähle $c \geq a$ (z.B. $c = a + 1$).
  Dann gilt für alle $x \in \mathbb{R}$: $|g(x)| = a \leq c \cdot 1$

- Beispiele
  - $3 \in O(1)$
  - $1.000.000 \in O(1)$
  - $O\left(\sqrt{313{,}56}\right) = O(1) = O(\pi)$

O-Notation Rechenregeln: Skalare Multiplikation

- Wenn $g \in O(f)$ gilt und $a \in \mathbb{R}$, dann ist

$$a \cdot g \in O(f)$$

- Beweis

  Es gibt $c > 0$ und $x_0 > 0$, sodass für alle $x \geq x_0$ gilt:

  $$|g(x)| \leq c \cdot |f(x)|$$

  Für $c' = c \cdot |a|$ gilt dann auch:

  $$|a \cdot g(x)| \leq c' \cdot |f(x)|$$

- Beispiele
  - $1.000\, n \in O(n)$
  - $23n^5 \in O(23n^5) = O(n^5)$
  - $2^{n+a} = 2^a \cdot 2^n \in O(2^n)$
  - $O(\log n) = O(\ln n) = O(\log_2 n)$

Basiswechsel: wegen $b = a^{\log_a b}$ gilt

$$a^{\log_a n} = n = b^{\log_b n} = a^{(\log_a b) \cdot \log_b n}$$

$$\log_a n = (\log_a b) \cdot \log_b n = const \cdot \log_b n$$

O-Notation Rechenregeln: Addition

- Wenn $g_1 \in O(f_1)$ und $g_2 \in O(f_2)$, dann gilt:

$$g_1 + g_2 \in O\big(\max(f_1, f_2)\big)$$

- Beweis

  Es gibt $c_1 > 0$, $c_2 > 0$ und $x_1 > 0$, $x_2 > 0$,
  sodass für alle $x \geq x_1$, $x \geq x_2$ gilt:
  $|g_1(x)| \leq c_1 \cdot |f_1(x)|$ sowie $|g_2(x)| \leq c_2 \cdot |f_2(x)|$
  Es gilt:

$$\begin{aligned} &|g_1(x) + g_2(x)| \leq |g_1(x)| + |g_2(x)| \\ &\leq c_1|f_1(x)| + c_2|f_2(x)| \leq 2 \cdot \max\left(c_1|f_1(x)|, c_2|f_2(x)|\right) \\ &\leq \max\left(2c_1|f_1(x)|, 2c_2|f_2(x)|\right) \end{aligned}$$

- Beispiele
  - $23n^5 + 2n^4 + 56n^3 + n^2 + 13n + 0.3 \in O(n^5)$
  - $\log n + n \in O(n)$

O-Notation Rechenregeln: Multiplikation

- Wenn $g_1 \in O(f_1)$ und $g_2 \in O(f_2)$, dann gilt:

$$g_1 \cdot g_2 \in O(f_1 \cdot f_2)$$

- Beweis

  Es gibt $c_1 > 0$, $c_2 > 0$ und $x_1 > 0$, $x_2 > 0$,
  sodass für alle $x \geq x_1$, $x \geq x_2$ gilt:
  $|g_1(x)| \leq c_1 \cdot |f_1(x)|$ und $|g_2(x)| \leq c_2 \cdot |f_2(x)|$

  Es gilt:

$$\begin{aligned} &|g_1(x) \cdot g_2(x)| = |g_1(x)| \cdot |g_2(x)| \\ &\leq c_1 \cdot |f_1(x)| \cdot c_2 \cdot |f_2(x)| \leq c_1 c_2 \cdot |f_1(x) f_2(x)| \end{aligned}$$

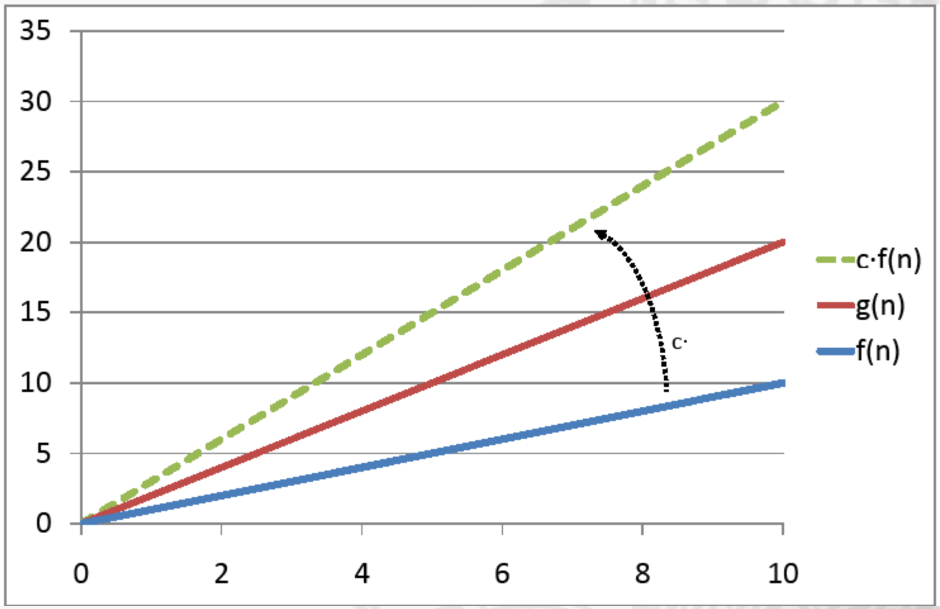
- Beispiele
  - $(\log n + n)\sqrt{n} \in O(n\sqrt{n})$
  - $O(2^n) \cdot O(2^m) = O(2^{n+m})$
  - $O(n \cdot \log\log n) \subset O(n \cdot \log n) \subset O(n^2) \subset O(n^2 \log n)$

Veranschaulichung (1)

- $O(f) = \{g: \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0: \exists n_0 > 0: \forall n \geq n_0: |g(n)| \leq c \cdot |f(n)|\}$

- $g(n) = 2n, f(n) = n \Rightarrow g \in O(f)$
- $c = 3,\ n_0$ beliebig $\Rightarrow g(n) = 2n \leq 3n = c \cdot f(n)$

- Konstante Faktoren werden vernachlässigt!

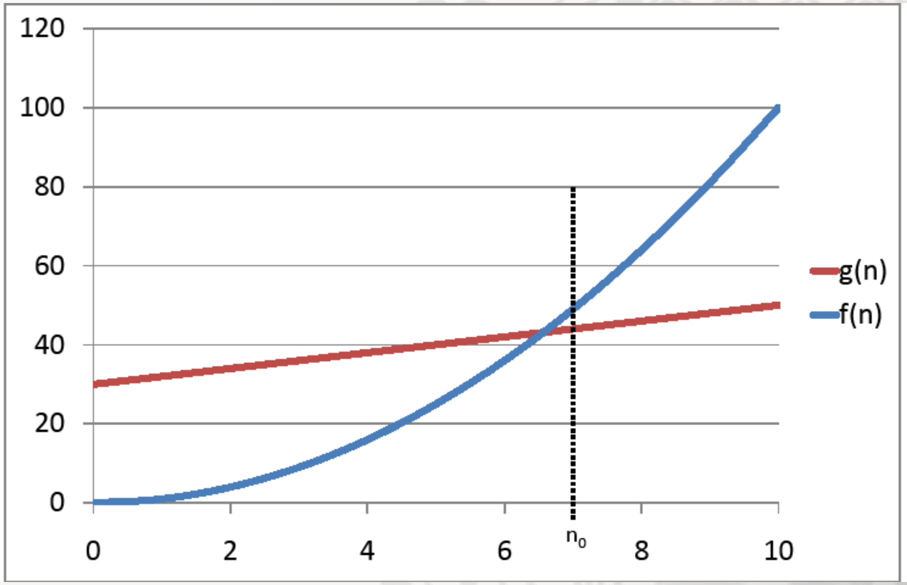


Veranschaulichung (2)

- $O(f) = \{g: \mathbb{N} \to \mathbb{R} \mid \exists c > 0: \exists n_0 > 0: \forall n \geq n_0: |g(n)| \leq c \cdot |f(n)|\}$

- $g(n) = 2n + 30, f(n) = n^2 \Rightarrow g \in O(f)$
- $c = 1,\ n_0 = 7 \Rightarrow g(n) \leq f(n)$ für alle $n \geq n_0 = 7$

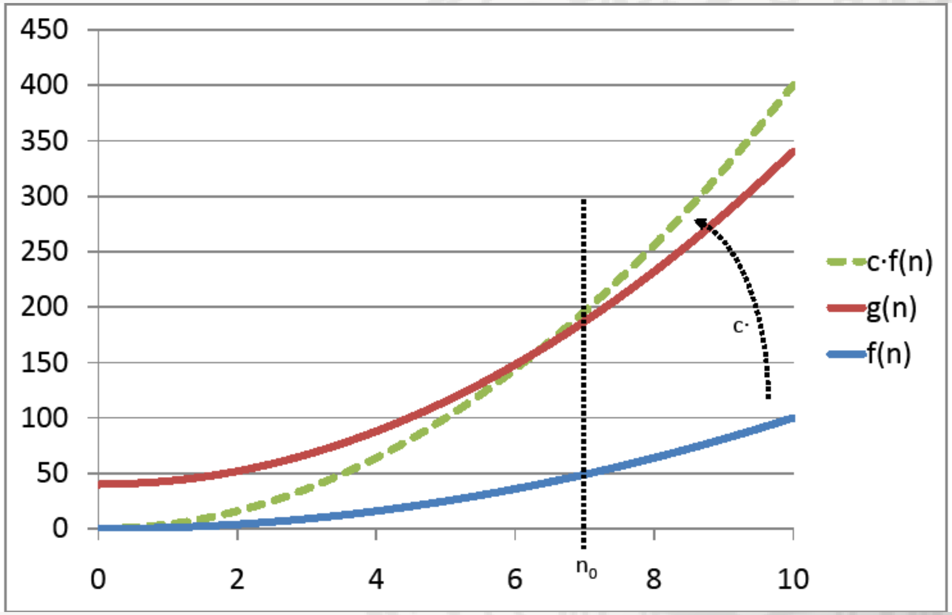
- für kleine n kann die „Laufzeit“ von $f(n)$ „besser“ sein
- bei asymptotischer Betrachtung wächst $g(n)$ jedoch langsamer!



Veranschaulichung (3)

- $O(f) = \{g: \mathbb{N} \to \mathbb{R} \mid \exists c > 0: \exists n_0 > 0: \forall n \geq n_0: |g(n)| \leq c \cdot |f(n)|\}$

- $g(n) = 3n^2 + 40, f(n) = n^2 \Rightarrow g \in O(f)$
- $c = 4,\ n_0 = 7 \Rightarrow g(n) \leq 4f(n)$ für alle $n \geq n_0 = 7$



Veranschaulichung (4)

- $O(f) = \{g: \mathbb{N} \to \mathbb{R} \mid \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : |g(n)| \leq c \cdot |f(n)|\}$

- $g(n) = 3^n, f(n) = 2^n \Rightarrow g \notin O(f)$
  - Angenommen, g $\in O(f)$ sei wahr.
  - Dann existiert $c > 0$ und ein $n_0 > 0$, so dass für alle n gilt:
  $$3^n \leq c \cdot 2^n$$
  - Mit anderen Worten: Der Grenzwert von $\frac{g(n)}{f(n)}$ existiert mit
  $$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{2}\right)^n = c < \infty$$
  - Widerspruch: Da $\lim_{n\to\infty} \left(\frac{3}{2}\right)^n = \infty$, existiert kein $c > 0$.

Komplexitätsklassen

Sei n die Länge der Eingabe (z.B. Arraylänge, Länge eines Strings)

Klasse	Bezeichnung	Beispiel	$N = 10$	$N = 100$
$O(1)$	konstant	Einzeloperation	1	1
$O(\log n)$	logarithmisch	Binäre Suche	4	7
$O(n)$	linear	Sequentielle Suche	10	100
$O(n \log^k n)$	quasilinear	Sortieren eines Arrays	40	700
$O(n^2)$	quadratisch	Matrixaddition	100	10.000
$O(n^3)$	kubisch	Matrixmultiplikation	1.000	1.000.000
$O(n^k)$	polynomiell			
$O(2^n)$	exponentiell	Edit-Distanz naiv	1.000	10^{30}
$O(n!)$	faktoriell	Permutationen	10^7	10^{158}
$O(n^n)$			10^{10}	10^{200}

Achtung: Diese Zahlenbeispiele berücksichtigen nicht c, n_0

Beziehungen zwischen Komplexitätsklassen

- Die Komplexität definiert eine totale Ordnung auf reell-wertigen Funktionen von reellen Zahlen. Es gilt zum Beispiel:
$$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^3)$$
- Insbesondere sind alle Potenzen von n bzgl. des Exponenten geordnet
$$\forall a \leq b\colon n^a \in O\left(n^b\right)$$
- Logarithmen unterschiedlicher Basen fallen in die gleiche Klasse:
$$\forall a, b > 1\colon\; O(\log_a n) = O(\log_b n)$$
- Der Logarithmus wächst langsamer als jede Potenz
$$\forall a > 0\colon \log n \in O(n^a)$$
$$\forall a > 0\colon n^a \notin O(\log n)$$
- Exponentialfunktionen wachsen superpolynomiell:
$$\forall a, b > 1\colon n^a \in O(b^n)$$
$$\forall b > 1\colon b^n \notin O(n^a)$$

Anwendung auf Algorithmen (1)

- Elementare Anweisungen sind $O(1)$
  - Variablendeklaration: `String str;`
  - Initialisierung und Zuweisungen: `int i = j + 3;`
  - Vergleiche: `if (j == 7)` …

- Laufzeiten für Sequenzen von Anweisungen werden addiert
  - Im Allgemeinen mehrere Codezeilen

```
System.out.println("Text eingeben:");
try {
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader in = new BufferedReader(isr);
  String s = in.readLine();
  System.out.println("Der eingelesene Text lautet: " + s);
} catch(IOException ex) {
  System.out.println(ex.getMessage());
}
```

Anwendung auf Algorithmen (2)

- Bei Sequenzen und Verzweigungen wird addiert $O(f) + O(g)$
  - Folge von Anweisungen: `x = 0; y = 0; z = 0;`
  - Bedingte Anweisungen: `if (a%2 == 0) a = a/2; else a++;`
  - Fallunterscheidungen: `switch(n) case: … break;`

- Schleifen werden multipliziert
  - Wenn die Anzahl der Schleifendurchläufe durch $O(f)$ beschränkt ist und der Schleifenrumpf maximal den Aufwand $O(g)$ hat, dann ist die Komplexitätsklasse der Schleife $O(fg)$.

```
boolean containsDuplicates(int[] values) {
  for(int i = 0; i < values.length; i++) {
    for (int j = 0; j < values.length; j++) {
      if (i == j) { continue; }
      if (values[i] == values[j]) { return true; }
    }
  }
  return false;
}
```

  - (`values.length` $- 0) \in O(n)$, damit ist containsDuplicates quadratisch

- Für Rekursionen ist es ein bisschen komplizierter

Fibonacci-Folge

- Fibonacci, 1202 n.Chr. (ital. Mathematiker):
  - berühmte Kaninchen-Aufgabe:
    - Start: 1 Paar Kaninchen
    - Jedes Paar wirft nach 2 Monaten ein neues Kaninchenpaar
    - dann monatlich jeweils ein weiteres Paar
    - Wie viele Kaninchenpaare gibt es nach einem Jahr, wenn keines der Kaninchen vorher stirbt?
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

- Anzahl im n-ten Monat lässt sich durch rekursive Funktion beschreiben:

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{falls } n > 1 \end{cases}$$

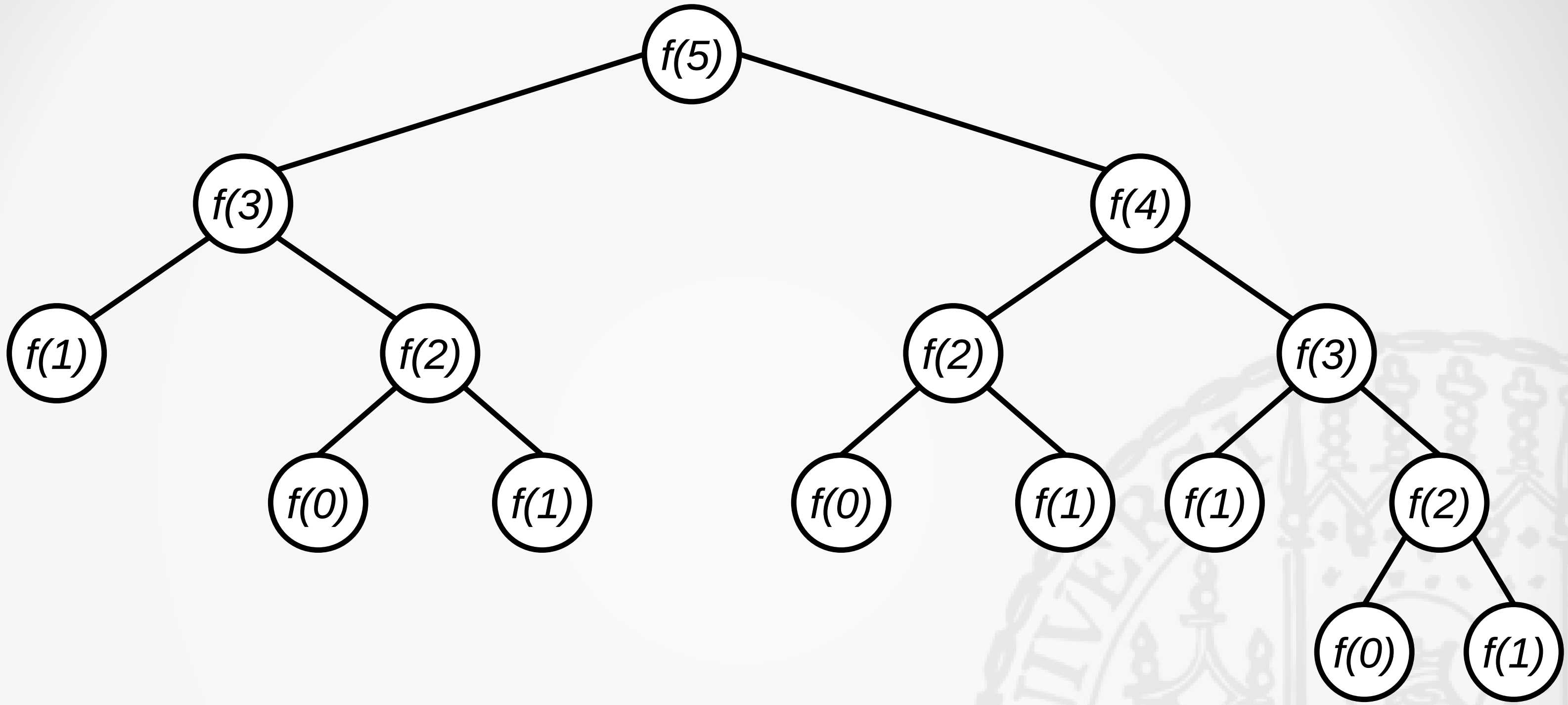
Naiver, rekursiver Algorithmus

- Definition direkt in ein Programm übertragen

$$\text{fib}(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{falls } n > 1 \end{cases}$$

```
int fib (int n) {
  if(n == 0)
    return 0;
  if(n == 1)
    return 1;
  return fib (n-1) + fib (n-2);
}
```

Berechnungsbaum für fib (5)



- Laufzeit liegt in $O(2^n)$ → bessere Laufzeit möglich?
- viele Aufrufe treten mehrfach auf
  (Frage: wie viele verschiedene Aufrufe gibt es?)

Endständige Rekursion

- Idee: Führe Ergebnis rekursiv mit
  - n zählt Schritte bis zum Ende
  - Rekursionsaufruf ist letzte Aktion; kein Nachklappern!
  - alte Werte auf dem Stack werden nicht mehr benötigt.

```
# Python
def fib(n, result=1, previous=0):
  if n == 0:
    return previous
  return fib(n - 1, result + previous, result)
```

- Endständige Rekursion ist Übergang zu einem iterativen Algorithmus

Ablaufprotokoll
für Aufruf fib(6):

n	result	prev.
6	1	0
5	1	1
4	2	1
3	3	2
2	5	3
1	8	5
0	13	8

Endständige Rekursion

- engl.: tail recursion
- Rekursiver Aufruf muss letzte Aktion sein.
- Ohne Endrekursion:
  - Zustand der Methode (Variablenbelegungen, Wiedereinsprungpunkt) muss als Stack Frame gespeichert werden → hoher Speicherplatzverbrach
- Vorteil Endrekursion:
  - Alle lokalen Variablen können verworfen werden
  - Rücksprung nicht mehr nötig
  - Stack Frames werden unnötig

  → Compiler kann endständige Rekursionen zu Iterationen optimieren

Fibonacci als iterativer Algorithmus

- Keine redundanten Berechnungen

```
int fib_iterative(n) {                // Start Fibonacci
  result = 0, previous = 1;           // Initialisiere Ergebnis-/Vorvariable
  while (n > 0) {                     // Schleife über Zahlen bis n rückwärts
    pprev = previous;                 // Merke Vorvoriges
    previous = result;                // Merke Voriges
    result = result + pprev;          // Aktuellen Wert berechnen
    n--;                              // Zähler runtersetzen
  }                                   // Schleifenende; Ergebnis berechnet
  return result;                      // Ergebnis zurückgeben
}                                     // Ende Fibonacci
```

- Laufzeitanalyse
  - Linear: $T(n) \in O(n)$
  - Enorme Verbesserung gegenüber (naiver) rekursiver Variante

Rekursion vs. Iteration

- Vergleich
  - Rekursive Formulierung oft eleganter
  - Iterative Lösung oft effizienter aber komplizierter

- Äquivalenz der Programmierprinzipien
  - Jede rekursive Lösung iterativ (d.h. mit Schleifen) lösbar und umgekehrt
  - Rekursion lässt sich in Spezialfällen automatisiert auflösen
    - endständige Rekursion

Analyse von Rekursionsgleichungen: Sukzessives Einsetzen

```
int fib (int n) {
  if(n <= 1) return n;
  else return fib(n-1) + fib(n-2);}
```

- Komplexität des iterativen Algorithmus: $T_{\text{iter}}(n) \in O(n)$
- Komplexität des rekursiven Algorithmus:

  $T_{rek}(n) = T_{rek}(n-1) + T_{rek}(n-2) + 2$ mit $T_{rek}(0) = T_{rek}(1) = 1$
- Vereinfachung, erlaubt wg. Monotonie: $T_{rek}(n) < 2 \cdot T_{rek}(n-1) + 2$
- Sukzessives Einsetzen:

$$
\begin{aligned}
T_{rek}(n) &< 2 \cdot T_{rek}(n-1) + 2 \\
&< 2 \cdot (2 \cdot T_{rek}(n-2) + 2) + 2 = 4 \cdot T_{rek}(n-2) + 6 \\
&< 4 \cdot (2 \cdot T_{rek}(n-3) + 2) + 6 = 8 \cdot T_{rek}(n-3) + 14 \\
&< 8 \cdot (2 \cdot T_{rek}(n-4) + 2) + 14 = 16 \cdot T_{rek}(n-4) + 30 \\
&< \cdots \\
&< 2^n \cdot T_{rek}(n-n) + (2^{n+1} - 2) \in O(2^n)
\end{aligned}
$$

Weitere Beispiele zum sukzessiven Einsetzen

- $T(1) = 1$ und $T(n) = T(n-1) + n$ für $n > 1$
  $$T(n) = T(n-1) + n = T(n-2) + (n-1) + n$$
  $$= \cdots = T(1) + 2 + \cdots + (n-2) + (n-1) + n$$
  $$= \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in O(n^2)$$
- $T(1) = 0$ und $T(n) = T(n/2) + n$ für $n > 1$
  $$T(n) = T\left(\frac{n}{2}\right) + n$$
  $$= T\left(\frac{n}{4}\right) + \frac{n}{2} + n$$
  $$= T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$$
  $$= \cdots$$
  $$= T(1) + \cdots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n$$
  $$= n \sum_{i=0}^{n} \frac{1}{2^i} = n \cdot 2 \cdot \left(1 - \frac{1}{2^{n+1}}\right) = 2n - \frac{n}{2^n} \in O(n)$$
  Runden, falls n keine Zweierpotenz ist.

Master-Theoreme

- Einsetzmethode ist aufwändig.
- Zur Analyse von Rekursionsgleichungen wäre eine Werkzeugbox aus universellen Formeln hilfreich.
- Die folgenden Mastertheoreme schlussfolgern obere Laufzeitschranken für rekursive Funktionen.
- Achtung: Für eine rekursive Funktion muss erst die passende Form gefunden werden, damit das richtige Mastertheorem angewendet wird.

Master-Theorem (Divide-and-Conquer)

Für Rekursionsgleichungen der Form

$$T(n) \leq \begin{cases} c & , n \leq 1 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & , n > 1 \end{cases}$$

mit $c > 0,\ a > 0,\ b > 1,\ f(n) \in O(n^d),\ d \geq 0$ gilt:

$$T(n) \in \begin{cases} O(n^d) & , d > \log_b a \\ O(n^d \log n) & , d = \log_b a \\ O(n^{\log_b a}) & , d < \log_b a \end{cases}$$

- Statt $T\left(\frac{n}{b}\right)$ auch $T\left(\left\lceil\frac{n}{b}\right\rceil\right)$ oder $T\left(\left\lfloor\frac{n}{b}\right\rfloor\right)$, falls b kein Teiler von n
- Weitere alternative Darstellungen existieren (vgl. Cormen et al.)

Master-Theorem (Divide-and-Conquer): Beispiel

Für Rekursionsgleichungen der Form

Sei $T(n) = T\left(\frac{n}{2}\right) + n$ und $T(1) = c_0$.

$$T(n) \leq \begin{cases} c & ,n \leq 1 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & ,n > 1 \end{cases}$$

$c = c_0$

$a = 1, b = 2,$
$f(n) \in O(n^1)$

mit $c > 0,\ a > 0,\ b > 1,\ f(n) \in O\left(n^d\right),\ d \geq 0$ gilt:

$$T(n) \in \begin{cases} O\left(n^d\right) & ,d > \log_b a \\ O\left(n^d \log n\right) & ,d = \log_b a \\ O\left(n^{\log_b a}\right) & ,d < \log_b a \end{cases}$$

$d = 1 > 0 = \log_2 1$
$\Rightarrow T(n) \in O(n)$

- Statt $T\left(\frac{n}{b}\right)$ auch $T\left(\left\lceil\frac{n}{b}\right\rceil\right)$ oder $T\left(\left\lfloor\frac{n}{b}\right\rfloor\right)$, falls b kein Teiler von n
- Weitere alternative Darstellungen existieren (vgl. Cormen et al.)

Master-Theorem (Divide-and-Conquer): Beweis (1/3)

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) = a \cdot \left(a \cdot T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n)$$

$$= a^2 \cdot T\left(\frac{n}{b^2}\right) + a \cdot f\left(\frac{n}{b}\right) + f(n) = \cdots$$

$$= a^{\log_b n} \cdot T\left(\frac{n}{b^{\log_b n}}\right) + a^{(\log_b n)\,-1} \cdot f\left(\frac{n}{b^{(\log_b n)-1}}\right) + \cdots + a \cdot f\left(\frac{n}{b}\right) + f(n)$$

$$= a^{\log_b n} \cdot T(1) + \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$a^{\log_b n} = b^{(\log_b a)\cdot(\log_b n)}$$
$$= b^{(\log_b n)\cdot(\log_b a)} = n^{\log_b a}$$

$$= n^{\log_b a} \cdot T(1) + \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\leq O\left(n^{\log_b a}\right) + \sum_{i=0}^{\log_b n-1} a^i O\left(\left(\frac{n}{b^i}\right)^d\right) = O\left(n^{\log_b a}\right) + O\left(n^d\right) \cdot \sum_{i=0}^{\log_b n-1} \left(\frac{a}{b^d}\right)^i$$

Master-Theorem (Divide-and-Conquer): Beweis (2/3)

Falls $\frac{a}{b^d} \neq 1$ folgt mit geom. Summenformel:

$$O\left(n^{\log_b a}\right) + O\left(n^d\right) \cdot \sum_{i=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^i$$

$$= O\left(n^{\log_b a}\right) + O\left(n^d\right) \cdot \frac{1 - \left(\frac{a}{b^d}\right)^{\log_b(n)-1+1}}{1 - \frac{a}{b^d}}$$

$$= O\left(n^{\log_b a}\right) + O\left(n^d\right) \cdot \frac{1 - n^{\log_b\left(\frac{a}{b^d}\right)}}{1 - \frac{a}{b^d}}$$

$$= O\left(n^{\log_b a}\right) + O\left(n^d\right) \cdot \frac{1 - \left(\frac{n^{\log_b a}}{n^d}\right)}{1 - \frac{a}{b^d}}$$

$$\leq O\left(n^{\log_b a}\right) + O\left(n^d\right) \cdot O\left(\frac{n^{\log_b a}}{n^d}\right) = O\left(n^{\log_b a}\right)$$

Erinnerung geom. Summenformel:
$\sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q}$,
falls $q \neq 1$

Wie zuvor:
$n^{\log_b x} = x^{\log_b n}$

O-Notation ignoriert Faktoren und Konstanten

Master-Theorem (Divide-and-Conquer): Beweis (3/3)

- Fall: $\log_b a > d$

  Bereits gezeigt: $T(n) \in O(\mathrm{n}^{\log_b a})$

- Fall: $\log_b a < d$

  $T(n) \in O(n^{\log_b a}) \subseteq O(n^d)$

- Fall: $\log_b a = d$

$$O(n^{\log_b a}) + O\left(n^d\right) * \sum_{i=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^i$$
$$= O(n^d) + O\left(n^d\right) * \sum_{i=0}^{\log_b(n)-1} 1^i$$
$$= O(n^d) + O\left(n^d\right) * \log_b n = O\left(n^d \log n\right)$$

Geom. Summenformel nicht anwendbar!

Master-Theorem (Subtract-and-Conquer)

Für Rekursionsgleichungen der Form

$$T(n) \leq \begin{cases} c & ,n \leq 1 \\ a \cdot T(n-b) + f(n) & ,n > 1 \end{cases}$$

mit $c > 0$, $a > 0$, $b > 0$, $f(n) \in O\left(n^d\right)$, $d \geq 0$ gilt:

$$T(n) \in \begin{cases} O\left(n^d\right) & ,a < 1 \\ O\left(n^{d+1}\right) & ,a = 1 \\ O\left(n^d a^{n/b}\right) & ,a > 1 \end{cases}$$

Master-Theorem (Subtract-and-Conquer): Beispiel

Fibonacci:
Sei $T(n) = T(n-2) + T(n-1)$
und $T(1) = 1$.

Für Rekursionsgleichungen der Form

$$T(n) \leq \begin{cases} c & ,n \leq 1 \\ a \cdot T(n-b) + f(n) & ,n > 1 \end{cases}$$

Wir „verschlimmern" die Laufzeit etwas:

mit $c > 0,\ a > 0,\ b > 0,\ f(n) \in O\left(n^d\right),\ d \geq 0$ gilt: $T(n) \leq 2T(n-1)$

$$T(n) \in \begin{cases} O\left(n^d\right) & ,a < 1 \\ O\left(n^{d+1}\right) & ,a = 1 \\ O\left(n^d a^{n/b}\right) & ,a > 1 \end{cases}$$

Mit $c = 1, a = 2, b = 1, d = 0$
folgt Fall 3: $T(n) \in O\left(n^d a^{\frac{n}{b}}\right) =$
$O\left(n^0 \cdot 2^{\frac{n}{1}}\right) = O(2^n)$

Master-Theorem (Subtract-and-Conquer): Beweis

$$T(n) = aT(n-b) + f(n) = a\big(aT(n-2b) + f(n-b)\big) + f(n)$$

$$= a^2T(n-2b) + af(n-b) + f(n) = \cdots = a^{n/b}T(0) + \sum_{i=0}^{n/b} a^i f(n-ib)$$

$$\leq O(a^{n/b}) + \sum_{i=0}^{n/b} a^i O\left((n-ib)^d\right)$$

$$= O(a^{n/b}) + O\left(n^d\right) \sum_{i=0}^{n/b} a^i$$

Für $a \neq 1$ gilt die geom. Summenformel:

$$\sum_{i=0}^{n/b} a^i = \frac{1-a^{n/b-1}}{1-a} \in O\left(a^{n/b}\right)$$

$$= \begin{cases} O\left(a^{n/b}\right) + O\left(n^d\right) \cdot O(1) = O\left(n^d\right), & a < 1 \\ O\left(a^{n/b}\right) + O\left(n^d\right) \cdot O(n) = O\left(n^{d+1}\right), & a = 1 \\ O\left(a^{n/b}\right) + O\left(n^d\right) \cdot O\left(a^{n/b}\right) = O\left(n^d a^{n/b}\right), & a > 1 \end{cases}$$

Substitution

- Idee: Schwierige Ausdrücke auf Bekanntes zurückführen
- Bsp: $T(n) = 2 \cdot T\left(\left\lfloor\sqrt{n}\right\rfloor\right) + \log_2 n$
  - Substituiere $m = \log_2 n \Leftrightarrow 2^{\mathrm{m}} = \mathrm{n}$
  - Ergibt: $T(2^m) = 2 \cdot T\left(2^{m/2}\right) + m$
  - Setze $S(m) = T(2^m)$
  - Ergibt: $S(m) = 2 \cdot S(m/2) + m$
  - Mit Mastertheorem: $S(m) \in O(m \log m)$
  - Rücksubstitution:
    $T(n) = T(2^m) = S(m) \in O(m \log m) = O(\log n \cdot \log \log n)$

Zusammenfassung Komplexität

- Analyse des Resourcenverbrauchs von Algorithmen (Laufzeit, Speicherbedarf) nicht im Detail, sondern als Komplexitätsklasse
- Dazu O-Notation als obere Schranken
- Analyse von Algorithmen geht direkt für einfache Anweisungen und Schleifen
- Für rekursive Funktionen schwieriger:
  - Sukzessives Einsetzen
  - Mastertheoreme
  - Substitution