

Kapitel 2: Effizienz und Komplexität

Effizienz und Laufzeiten

O-Notation

Rekursion

Effizienz von Algorithmen

- Effizienzfaktoren:
 - Rechenzeit (Anzahl der Einzelschritte)
 - Speicherplatzbedarf
 - Zugriffe auf Sekundärspeicher (z.B. Festplatte)
 - Kommunikationsaufwand (z.B. Netzwerk)
- konkrete Laufzeit eines Algorithmus hängt von vielen Faktoren ab
 - Takt der CPU
 - Länge der Eingabe
 - Implementierung der Basisoperationen
- axiomatisches Rechnermodell als Vergleichsmaßstab (eingeschränkt) möglich

Komplexität

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

Problem	$T(n)$
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 $(m \times n) * (n \times m)$	# Skalarmultiplikationen

- Üblicherweise asymptotische Betrachtung

Beispiel: Einfügen eines Stackelements

- Neue Liste initialisieren
 - Speicher für Pointer „stack“ allokkieren
 - Speicher für Liste allokkieren
 - Liste initialisieren
 - value = null, next = null
 - Listenpointer an „stack“ übergeben
 - value-Attribut setzen
 - next-Attribut setzen
 - first-Attribut setzen
-
- Alle Operationen benötigen konstant viel Aufwand, unabhängig vom einzufügenden Wert
 - Die Methode benötigt damit immer konstant viel Aufwand

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

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: $A \in n \times n$, $B \in n \times n$

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

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

In der Informatik liegen meist diskrete Eingaben vor:

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

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) // n Operationen
        summe += i;          // je 1 Operation
    return summe;            // 1 Operation
}
```

- Damit ergeben sich $1 + n \cdot 1 + 1$ Operationen

- Erinnerung:

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

- $g(n) = n + 2$

$$n + 2 \leq 2n \text{ für } c = 2, n_0 = 2 \text{ und } f(n) = n \Rightarrow g \in O(n)$$

- Es gibt unendlich viele Alternativen, c zu wählen:

$$n + 2 \leq 1001 \cdot n \text{ für } c = 1001, n_0 \geq \frac{1}{500} \text{ und } f(n) = n \Rightarrow g \in O(n)$$

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:
 - $3 \in O(1)$
 - $1.000.000 \in O(1)$
 - $O(\sqrt{313,56}) = O(1) = O(-25,4) = O(\pi)$

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

Basiswechselsatz: $\log_b n = \frac{\log_a n}{\log_a b}$
 $\Rightarrow O(\log_b n) \cdot O(\log_a b) = O(\log_a n)$
 $= O(\log_b n) \cdot \text{const} = O(\log_a n)$

O-Notation Rechenregeln: Addition

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

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

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:

$$|g_1(x)| \leq c_1 \cdot |f_1(x)| \text{ sowie } |g_2(x)| \leq c_2 \cdot |f_2(x)|$$

Es gilt:

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

- 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:

$$|g_1(x)| \leq c_1 \cdot |f_1(x)| \text{ und } |g_2(x)| \leq c_2 \cdot |f_2(x)|$$

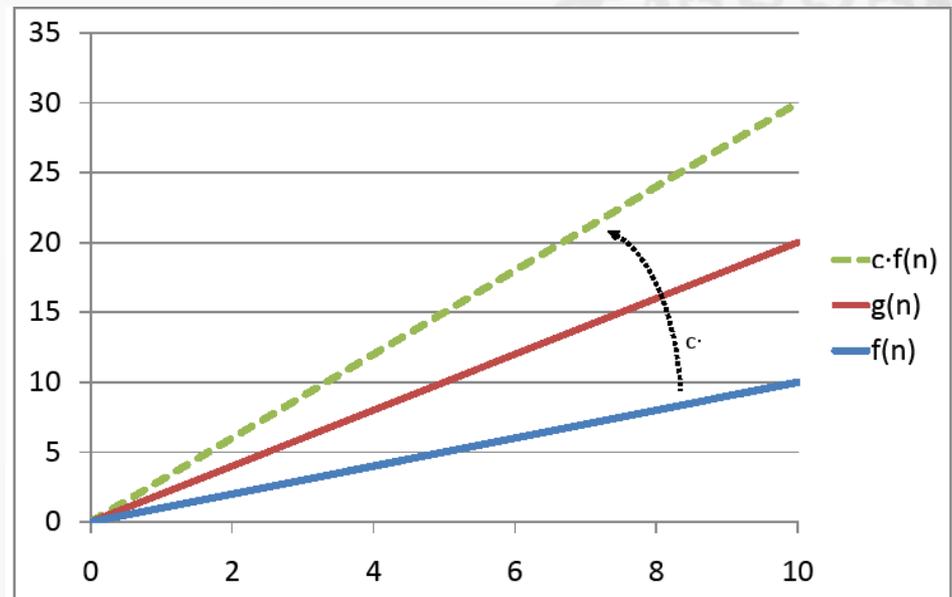
Es gilt:

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

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

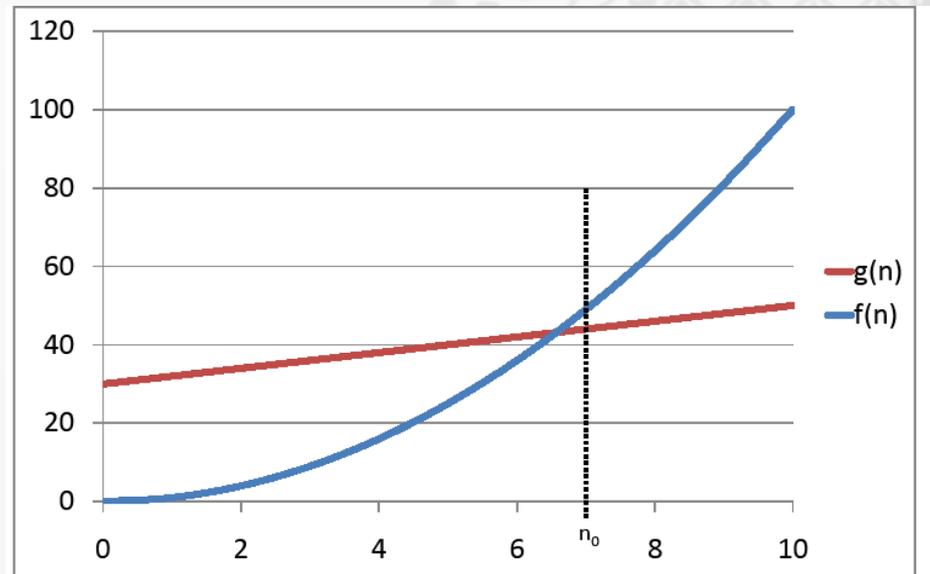
Veranschaulichung (1)

- $O(f) = \{g: \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0: |g(n)| \leq c \cdot |f(n)|\}$
- $g(n) = 2n, f(n) = n \Rightarrow g \in O(f)$
- $c = 3, n_0$ beliebig $\Rightarrow g(n) = 2n \leq 3n = c \cdot f(n)$
- Konstante Faktoren werden vernachlässigt!



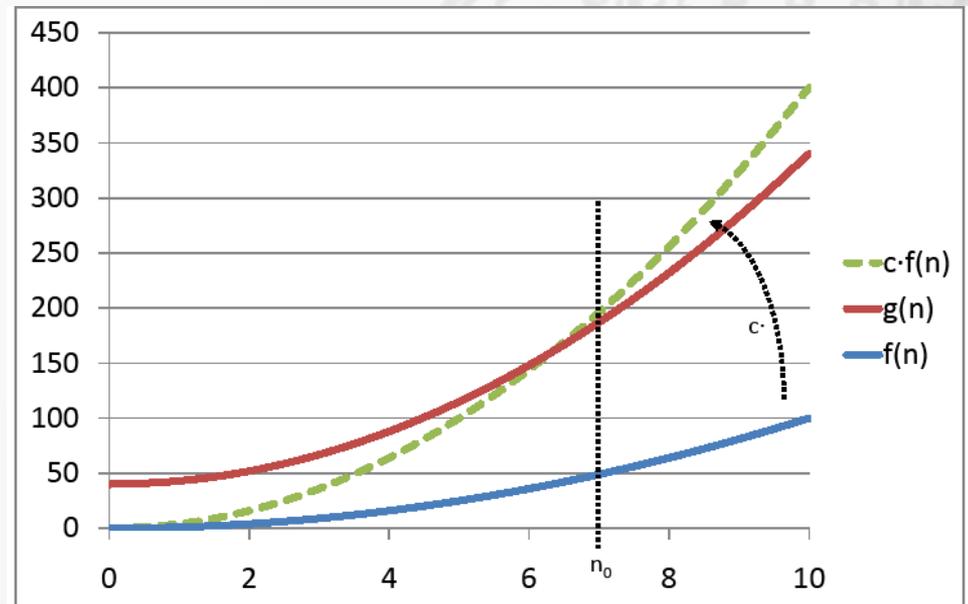
Veranschaulichung (2)

- $O(f) = \{g: \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0: |g(n)| \leq c \cdot |f(n)|\}$
- $g(n) = 2n + 30, f(n) = n^2 \Rightarrow g \in O(f)$
- $c = 1, n_0 = 7 \Rightarrow g(n) \leq f(n)$ 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)

- $O(f) = \{g: \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0: |g(n)| \leq c \cdot |f(n)|\}$
- $g(n) = 3n^2 + 40, f(n) = n^2 \Rightarrow g \in O(f)$
- $c = 4, n_0 = 7 \Rightarrow g(n) \leq 4f(n)$ für alle $n \geq n_0 = 7$



Veranschaulichung (4)

- $O(f) = \{g: \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0 \exists n_0 > 0 \forall n \geq n_0: |g(n)| \leq c \cdot |f(n)|\}$
- $g(n) = 3^n, f(n) = 2^n \Rightarrow g \notin O(f)$
 - Angenommen, $g \in O(f)$ sei wahr.
 - Dann existiert $c > 0$ und ein $n_0 > 0$, so dass für alle n gilt:
$$3^n \leq c2^n$$
 - Mit anderen Worten: Der Grenzwert von $\frac{g(n)}{f(n)}$ existiert mit
 - $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{3^n}{2^n} = \lim_{n \rightarrow \infty} \left(\frac{3}{2}\right)^n = c < \infty$
 - Widerspruch: Da $\lim_{n \rightarrow \infty} \left(\frac{3}{2}\right)^n = \infty$, existiert kein $c > 0$.

Komplexitätsklassen

Sei n die Länge der Eingabe (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, n_0

Beziehungen zwischen Komplexitätsklassen

- Die Komplexität definiert eine Totalordnung auf reellwertigen 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

$$\forall a \leq b: n^a \in O(n^b)$$

- Logarithmen unterschiedlicher Basen fallen in die gleiche Klasse:

$$\forall a, b > 1: O(\log_a n) = O(\log_b n)$$

- Der Logarithmus wächst langsamer als jede Potenz

$$\forall a > 0: \log n \in O(n^a)$$

$$\forall a > 0: n^a \notin O(\log n)$$

- Exponentialfunktionen wachsen superpolynomiell:

$$\forall a, b > 1: n^a \in O(b^n)$$

$$\forall b > 1: b^n \notin O(n^a)$$

Anwendung auf Programmcode (1)

- Elementare Anweisungen sind $O(1)$
 - Variablendeklaration: `String str;`
 - Initialisierung und Zuweisungen: `int i = j + 3;`
 - Vergleiche: `if(name == „Fibonacci“) ...`
- 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 Programmcode (2)

- Verzweigungen werden addiert $O(f) + O(g)$
 - Bedingte Blöcke: `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;
}
```
 - $(\text{values.length} - 0) \in O(n)$, damit ist `containsDuplicates` quadratisch
- Rekursionen und Seiteneffekte machen es nun kompliziert

Fibonacci-Reihe

- 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:

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

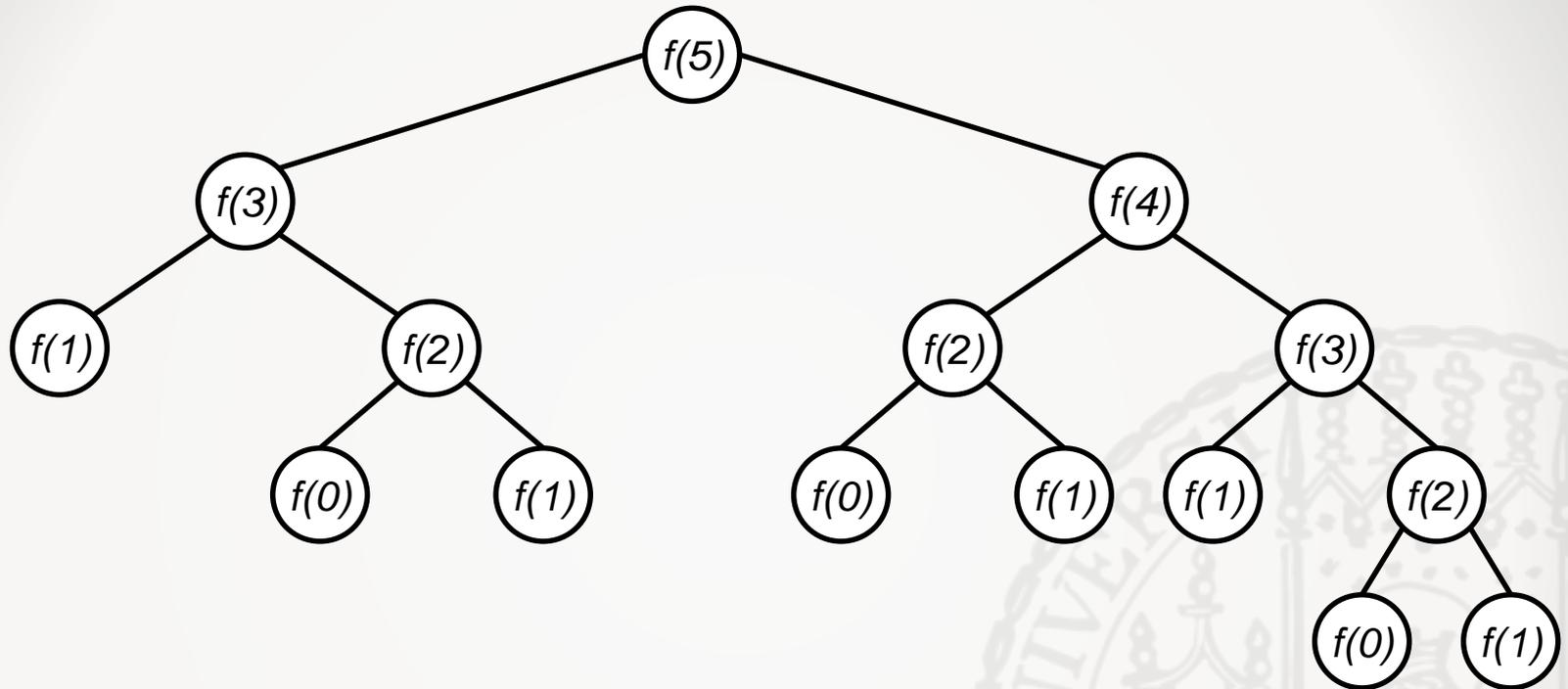
Naiver, rekursiver Algorithmus

- 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)



- wir zeigen gleich (p.27): Laufzeit liegt in $O(2^n)$
→ bessere Laufzeit möglich?
- viele Aufrufe treten mehrfach auf

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

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 Speicherplatzverbrauch
- 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 ist nicht immer schlecht: endständige Rekursion

Analyse von Rekursionsgleichungen: Sukzessives Einsetzen

- Komplexität des iterativen Algorithmus: $T_{\text{iter}}(n) \in O(n)$
- Komplexität des rekursiven Algorithmus: $T_{\text{rek}}(n) \in O(2^n)$

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

$$\begin{aligned} T_{\text{rek}}(n) &= T_{\text{rek}}(n-1) + T_{\text{rek}}(n-2) + O(1) \\ &< 2T_{\text{rek}}(n-1) + O(1) \\ &< 2(2T_{\text{rek}}(n-2) + O(1)) + O(1) \\ &= 4T_{\text{rek}}(n-2) + 2O(1) + O(1) \\ &< 8T_{\text{rek}}(n-3) + 4O(1) + 2O(1) + O(1) \\ &< \dots \\ &< 2^n T_{\text{rek}}(n-n) + 2^{n-1}O(1) + \dots + 2O(1) + O(1) \\ &= 0 + \sum_{i=0}^{n-1} 2^i \cdot O(1) \stackrel{\text{geom. Reihe}}{=} \frac{2^n - 1}{2 - 1} \cdot O(1) = (2^n - 1) \cdot O(1) \in O(2^n) \end{aligned}$$

Weitere Beispiele zum sukzessiven Einsetzen

- $T(1) = 1$ und $T(n) = T(n - 1) + n$ für $n > 1$
 $T(n) = T(n - 1) + n = T(n - 2) + (n - 1) + n$
 $= \dots = T(1) + 2 + \dots + (n - 2) + (n - 1) + n$
 $= \sum_{i=1}^n i = \frac{n(n + 1)}{2} \in O(n^2)$

- $T(1) = 0$ und $T(n) = T(n/2) + n$ für $n > 1$
 $T(n) = T\left(\frac{n}{2}\right) + n$
 $= T\left(\frac{n}{4}\right) + \frac{n}{2} + n$
 $= T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n$
 $= \dots$
 $= T(1) + \dots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n$
 $= n \sum_{i=0}^{\log_2 n} \frac{1}{2^i} = n \cdot 2 \cdot \left(1 - \frac{1}{2^{\log_2 n + 1}}\right) = 2n - \frac{n}{2^{\log_2 n}} \in O(n)$

Annahme: n ist 2er-Potenz

Master-Theoreme

- Einsetzmethode ist aufwendig.
- 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 $T\left(\left\lfloor\frac{n}{b}\right\rfloor\right)$ oder $T\left(\left\lceil\frac{n}{b}\right\rceil\right)$, 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 $T(1) = c_0$.

$$T(n) \leq \begin{cases} c & , n \leq 1 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & , n > 1 \end{cases} \quad \begin{array}{l} c = c_0 \\ a = 1, b = 2, \\ f(n) \in O(n^1) \end{array}$$

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

$$T(n) \in \begin{cases} O(n^d) & , d > \log_b a \\ O(n^d \log n) & , d = \log_b a \\ O(n^{\log_b a}) & , d < \log_b a \end{cases} \quad \begin{array}{l} d = 1 > 0 = \log_2 1 \\ \Rightarrow T(n) \in O(n) \end{array}$$

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

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

Erinnerung zum Logarithmus:

$$\log_b b^x = b^{\log_b x} = x$$

$$\log_b x^k = k \log_b x$$

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

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

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

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

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

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

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

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

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

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

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

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

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

- Fall: $\log_b a < d$
 $T(n) \in O(n^{\log_b a}) \subseteq O(n^d)$

- Fall: $\log_b a = d$

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

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

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

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

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

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

Fibonacci:

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

Wir „verschlimmern“ die
Laufzeit etwas:

$$T(n) \leq 2T(n - 1)$$

Mit $c = 1$, $a = 2$, $b = 1$, $d = 0$
folgt Fall 3:

$$T(n) \in O(n^d a^{n/b}) = O(2^n)$$

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

$$\begin{aligned} T(n) &= aT(n-b) + f(n) = a(aT(n-2b) + f(n-b)) + f(n) \\ &= a^2T(n-2b) + af(n-b) + f(n) = \dots = a^{n/b}T(0) + \sum_{i=0}^{n/b} a^i f(n-ib) \end{aligned}$$

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

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

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

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

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

Substitution

- Idee: Schwierige Ausdrücke auf Bekanntes zurückführen
- Bsp: $T(n) = 2 \cdot T(\lfloor \sqrt{n} \rfloor) + \log n$
 - Substituiere $m = \log n$
 - Ergibt: $T(2^m) = 2 \cdot T(2^{m/2}) + m$
 - 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:
 $T(n) = T(2^m) = S(m) \in O(m \log m) = O(\log n \cdot \log \log n)$