Überblick

1. Die „Prozedur“ als Kernbegriff der Programmierung
2. Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen
3. Prozeduren versus Prozesse
4. Ressourcenbedarf – Größenordnungen
5. Beispiel: Der größte gemeinsame Teiler

Notwendigkeit der Rekursion (1)

- NUR Rekursion ermöglicht in einer rein funktionalen Programmiersprache eine Anzahl von Wiederholungen, die von aktuellen Parametern abhängt.
- Betrachte:

```
summe(0) = 0
summe(n) = n + … + 0 falls n ∈ N und n ≥ 1
```

- In SML mit Pattern Matching:

```
fun summe(0) = 0
  | summe(n) = n + summe(n-1);
```

Notwendigkeit der Rekursion (2)

- Ohne Rekursion müsste man die Anzahl der Summanden unabhängig von dem formalen Parameter `n` festlegen:

```
fun summe'(0) = 0
  | summe'(1) = 1 + 0
  | summe'(2) = 2 + 1 + 0
  | summe'(3) = 3 + 2 + 1 + 0
  | summe'(4) = 4 + 3 + 2 + 1 + 0
  | ???
```

Diese Deklaration ist unvollständig und inkorrekt!

Rekursion versus Iteration

- Algorithmen, die Wiederholungen mittels Schleifen statt Rekursion spezifizieren, werden „*iterativ*“ genannt.
- Die Technik, auf der iterative Algorithmen beruhen, heißt „*Iteration*“.
- Dazu 3 Beispiele:

  Zeichne ein Viereck auf den Boden mittels
  - Rekursion
  - Iteration in Form einer `for` – Schleife
  - Iteration in Form einer `while` – Schleife

Viereckalgorithmus mit Rekursion

Zeichne eine Seite wie folgt:
Gehe 3 Schritte nach vorn und zeichne dabei eine Linie; Wende dich um 90 Grad nach rechts.
Wenn du nicht am Startpunkt stehst, dann zeichne eine Seite unter Anwendung des oben geschilderte Verfahrens.

- Die Rekursion ist bereits bekannt.

Viereckalgorithmus mit Iteration (`for` – Schleife)

Wiederhole 4 Mal:
Gehe 3 Schritte nach vorn und zeichne dabei eine Linie; Wende Dich um 90 Grad nach rechts.

- Die Anzahl der Wiederholungen ist explizit festgelegt.

Viereckalgorithmus mit Iteration (`while` – Schleife)

Gehe 3 Schritte nach vorn und zeichne dabei eine Linie; wende Dich um 90 Grad nach rechts.
Wiederhole dies, solange Du nicht am Startpunkt stehst.

- Die Anzahl der Wiederholungen hängt von einer Bedingung ab, die nach jeder Wiederholung ausgewertet werden muss.

Rekursion versus Iteration: Komplexitätsaspekte am Beispiel

- Betrachte die über den natürlichen Zahlen totale Fakultätsfunktion:

Diese Definition ist ziemlich unpräzise.

$0! = 1$

$n! = n * (n-1) * \ldots * 1$ falls $n \in N$ und $n \geq 1$

Diese Definition ist für uns besser geeignet.

$0! = 1$

$n! = n * (n-1)!$ falls $n \in N$ und $n \geq 1$

- In SML:

```
fun fak(0) = 1
  | fak(n) = n * fak(n - 1);
```

Rekursive Auswertung von `fak`

- Auswertung von `fak(4)` mit dem Substitutionsmodell:

```
fak(4)
4 * fak(3)
4 * (3 * fak(2))
4 * (3 * (2 * fak(1)))
4 * (3 * (2 * (1 * fak(0))))
4 * (3 * (2 * (1 * 1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

Diese Auswertung braucht Speicher für unvollendete Zwischenberechnungen (z.B. `4*(3*(2*fak(1)))`. Damit die unterbrochenen Zwischenberechnungen so bald wie möglich korrekt weitergeführt werden, ist eine Buchführung notwendig, die Zeit kostet.

Eine andere Auswertung von `fak`

- Eine „billigere“ Auswertung von `fak(4)`:

```
(*)   4           1
      4 - 1       1 * 4
      3           4
      3 - 1       4 * 3
      2           12
      2 - 1       12 * 2
      1           24
      1 - 1       24 * 1
      0           24
```

Es gibt keine unterbrochenen Zwischenberechnungen, dadurch ist die Buchführung einfacher. Es wird weniger Speicher gebraucht.

- Links steht der jeweilige Wert von n, rechts die Teilprodukte dazu.
- Rechts werden die Zwischenergebnisse „akkumuliert“.

Imperativer, iterativer `fak` – Algorithmus

- Führe die Schritte aus `(*)` imperativ aus:

```
(**) fun  fak_imperativ(x) =
          let
                val akk = ref 1;
                val n = ref x;
          in
                (
                  while (!n > 0)
                  do
                      (
                            akk := !akk * !n;
                            n := !n -1
                      );
                  !akk
                )
          end;
```


Rein funktionaler, iterativer `fak`-Algorithmus

- Führe die Schritte aus `(*)` funktional aus:

```
fun fak_funktional(n) =
          let fun hilf_fak_funktional (n,akk)=
                if n = 0
                then akk
                else hilf_fak_funktional (n-1,akk*n)
          in
                hilf_fak_funktional (n, 1)
          end
```

Auswertung von `fak_iter` mit dem Substiutionsmodell

- Überprüfe ob `fak_funktional` die Schritte aus `(*)` macht:

```
fak_funktional ( 4 )
hilf_fak_funktional (4, 1 )
hilf_fak_funktional (4 - 1, 1 * 4 )
hilf_fak_funktional (3, 4 )
hilf_fak_funktional (3 - 1, 4 * 3 )
hilf_fak_funktional (2, 12 )
hilf_fak_funktional (2 - 1, 12 * 2 )
hilf_fak_funktional (1, 24 )
hilf_fak_funktional (1 - 1, 24 * 1 )
hilf_fak_funktional (0, 24 )
24
```

- Diese rekursive Funktion implementiert den iterativen Alg. `(**)`.

Rekursion versus Iteration: Vergleich am Beispiel `fak` (1)

- Der Unterschied zwischen `fak_funktional` und `fak_imperativ` ist unwesentlich. Beide Formen sind ähnlich (un-)anschaulich.
- An diesem Beispiel scheint die Iteration der Rekursion überlegen, weil dieser Formalismus „natürlich“ zu dem effizienteren Algorithmus führt:
  - Unter Verwendung eines funktionalen Formalismus ist es natürlicher, die Fakultät einer natürlichen Zahl mit `fak` statt mit `fak_funktional` zu berechnen.
  - Unter Verwendung von Zustandsvariablen und der Iteration ist `fak_imperativ` sehr natürlich.

Rekursion versus Iteration: Vergleich am Beispiel `fak` (2)

- `fak` ist weniger effizient als `fak_imperativ` bzw. `fak_funktional`, aber viel einfacher zu entwickeln und zu warten.
- `fak` kann sehr leicht mittels des Substitutionsmodells überprüft werden. Für `fak_funktional` gilt das auch, für `fak_imperativ` nicht.
- Mit dem Substitutionsmodell lässt sich die totale Korrektheit auf den natürlichen Zahlen von `fak` sehr leicht beweisen.

So ähnlich können in vielen praktischen Fällen die Vorzüge von Iteration und Rekursion einander gegenüber gestellt werden. Jeder Ansatz hat seine Anhänger.

Endrekursion (1)

- Die Funktion `fak_funktional` (mit ihrer Hilfsfunktion `hilf_fak_funktional`) kommt zu denselben Berechnungen wie `(**)`, weil bei der Auswertung von `fak_funktional(n)` keine Zwischenberechnung unterbrochen wird.
- Sonst würde es zu anderen Berechnungen kommen.

Endrekursion (2)

- An der Syntax einer rekursiven Funktion erkennt man, ob ihre Auswertung eine Unterbrechung von Berechnungen verlangt:
  - Kommt der rekursive Aufruf `R` in einem zusammengesetzten Ausdruck `A` vor, der keine Fallunterscheidung (`if-then-else` oder `case`) ist, dann werden Berechnungen unterbrochen, weil die Auswertung von `A` erst dann möglich ist, wenn der zugehörige Teilausdruck `R` ausgewertet wurde.
  - Kommt der rekursive Aufruf nicht in einem zusammengesetzten Ausdruck vor, der keine Fallunterscheidung (`if-then-else` oder `case`) ist, dann sind keine Unterbrechungen von Berechnungen nötig.

Beispiel:
rekursiver Aufruf `hilf_fak_funktional(n - 1, akk * n)` im Rumpf der Funktion `hilf_fak_funktional`.

Endrekursion Definition

- Eine rekursive Funktion nennt man „*endrekursiv*“ (*tail recursive*), wenn ihr Rumpf keinen rekursiven Aufruf enthält, der ein echter Teilausdruck eines zusammengesetzten Ausdrucks `A` ist, so dass `A` weder ein `if-then-else`-Ausdruck noch ein `case`-Ausdruck ist (`if-then-else`-Ausdrücke und `case`-Ausdrücke sind Sonderausdrücke mit Auswertung durch Sonderalgorithmen!).
- Fast immer werden Fallunterscheidungen durch Basisfälle und Rekursionsfälle unterschieden, deshalb werden sie in der Definition der Endrekursion nicht wie herkömmliche Ausdrücke behandelt. Sonst würde die Definition alle rekursiven Definitionen ausschließen!

Endrekursive Funktionen und Prozeduren

- Endrekursive Funktionen sind rekursiv. Ihre Auswertung führt aber zu iterativen (Berechnungs-) Prozessen.
- Ist eine Prozedur (bzw. ein Programm) rekursiv oder endrekursiv, so meint man damit die Syntax der Prozedur (bzw. des Programms), nicht den (Berechnungs-) Prozess, der sich aus dieser Prozedur (bzw. Programm) ergibt.

Die Funktion `hilf_fak_funktional` ist also rein syntaktisch rekursiv, aber auch endrekursiv. Damit ist sichergestellt, dass der Berechnungsprozess, der von dieser Funktion ausgelöst wird, ein iterativer Prozess ist.

Lineare und quadratische Rekursion am Beispiel

- Zur Rekursion der Potenz `n` betrachte die „Fibonacci-Zahlen“:

  fib(0) = 0

  fib(1) = 1

  fib(n) = fib(n – 1) + fib(n – 2) für $n \in N$ und $n \geq 2$

- In SML:

```
fun fib(0) = 0
  | fib(1) = 1
  | fib(n) = fib(n-1) + fib(n-2);
```

Auswertungsdauer von `fib`

- Bis zu Werten wie z.B. `n = 20` erfolgt die Auswertung von `fib(n)` schnell.
- Für größere Werte wie z.B. `n = 40` wird die Rechenzeit für die Auswertung deutlich länger.
- Das liegt an der besonderer Form der Rekursion in der obigen Definition:

  Im definierenden Teil der Funktion `fib` kommen ZWEI rekursive Aufrufe vor.

Lineare, quadratische Rekursion, Rekursion der Potenz n

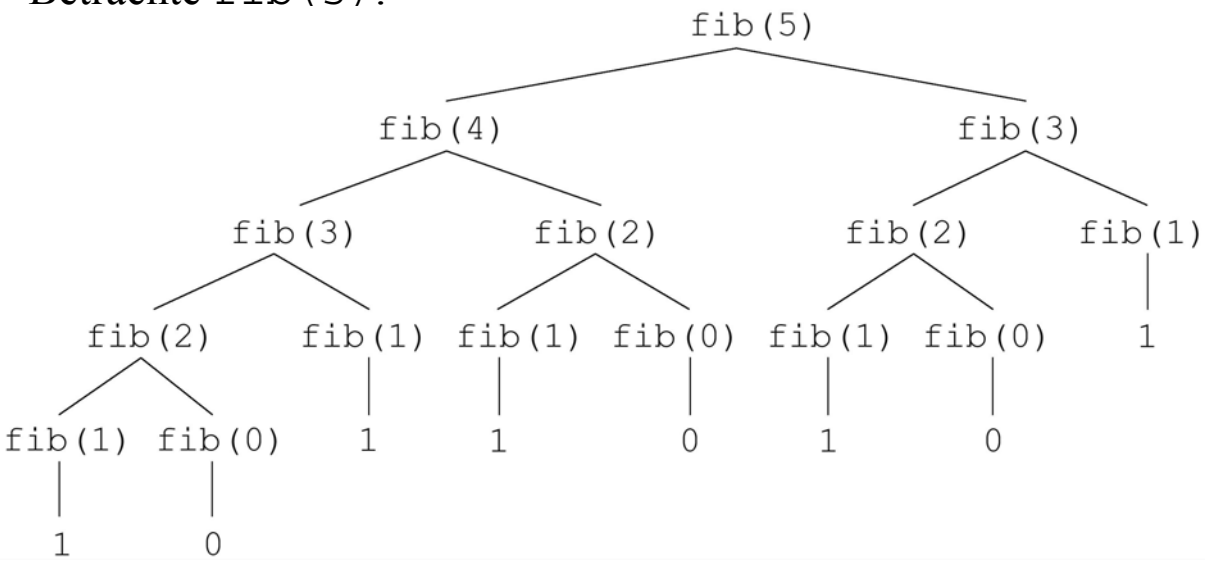
- Kommt nur ein rekursiver Aufruf im Rumpf einer Funktion vor (wie bei `fak` und `hilf_fak_funktional`), so heißt die Funktion „*linear rekursiv*“.
- Kommen zwei rekursive Aufrufe im Rumpf einer Funktion vor (wie bei `fib`), so heißt die Funktion „*quadratisch rekursiv*“.
- Kommen `n` rekursive Aufrufe im Rumpf einer Funktion vor, so heißt die Funktion „*rekursiv in der Potenz `n`*“.

!!!

Dass die Funktion `fib(n)` quadratisch rekursiv ist, heißt nicht, dass die benötigte Berechnungszeit für die Auswertung von `fib(n)` proportional zu n^2 ist! (Sie ist proportional zu `exp(n)`)

Baumrekursive Funktionen

- Rekursive Funktionen, die quadratisch rekursiv oder rekursiv in der Potenz n sind, heißen auch „*baumrekursiv*“.
- Betrachte `fib(5)`:



Einschub Goldener Schnitt

- Die Fibonacci-Funktion verhält sich asymptotisch wie die Funktion: $\mathbb{N} \to \mathbb{N}$

  $n \mapsto \Phi^n$
- Dabei ist Φ der „goldene Schnitt“, d.h. die Zahl, die durch die Gleichung $\Phi^2 = \Phi + 1$ definiert ist, d.h.:

  $\Phi = (1+\sqrt{5})/2 \approx 1{,}6180$
- Mit Hilfe dieses Zusammenhangs kann man zeigen, dass die Anzahl der Blätter im obigen Baum und damit die benötigte Berechnungszeit exponentiell mit n wächst.

Iterative Auswertung der baumrekursiven Funktion `fib` (1)

- Für baumrekursive Funktionen kann man oft iterative Auswertungen finden:

```
fun fib_imperativ(x) =
    let
       val i = ref 0;
       val akk1 = ref 0;(*!akk1: die i-te Fib-zahl*)
       val akk2 = ref 1;(*!akk2: die (i+1)-te Fib-zahl*)
       val n = ref x;
       val z = ref 0;
    in(
       while (!i < !n)
       do(
            i := !i + 1;
            z := !akk2;
            akk2 := !akk1 + !akk2;
            akk1 := !z
       );
       !akk1
    )
    end;
```

Für die Fibonaccizahlen sind zwei Akkumulatoren `akk1`, `akk2` und eine Zwischenvariable `z` nötig.

Iterative Auswertung der baumrekursiven Funktion `fib` (2)

- Die Variable `z` „rettet“ den alten Inhalt von `akk2`, bevor er verändert wird, damit der alte Inhalt anschließend zum Inhalt von `akk1` gemacht werden kann.
- Berechnung für `!n = 8`:

!i	!akk1	!akk2
0	0	1
1	1	1
2	1	2
3	2	3
4	3	5
5	5	8
6	8	13
7	13	21
8	21	34

Analog zu der Fakultätsfunktion (nur mit zwei Akkumulatoren) kann dieser iterative Berechnungsprozess auch mit einer endrekursiven Funktion erzielt werden.

Überlegung

- Eine Funktion wie `fib` verlangt wiederholt Auswertungen der gleichen Ausdrücke:
- Bei der Auswertung von `fib(5)` werden

  `fib(4)` 1 Mal
  `fib(3)` 2 Mal
  `fib(2)` 3 Mal
  `fib(1)` 5 Mal
  `fib(0)` 3 Mal ausgewertet.

- Gibt es eine effiziente Auswertung der baumrekursiven Funktion `fib`, die keine Veränderung ihrer Implementierung erfordert?

Memoisierung (1)

- Bei einer Veränderung des Auswertungsalgorithmus kann Buch darüber geführt werden, welche Aufrufe der Funktion `fib` (oder einer anderen Funktion) bereits ausgewertet worden sind:
  - Wurde ein rekursiver Aufruf `A` mit Ergebniswert `W` ausgewertet, so wird die Gleichung `A = W` in einer Tabelle `T` gespeichert („*memoisiert*“).
  - Soll ein rekursiver Aufruf `A` ausgewertet werden, so wird zunächst in der Tabelle `T` nach einem Eintrag `A = W` gesucht. Gibt es einen solchen Eintrag, so wird der Aufruf `A` nicht wieder ausgewertet, sondern der Wert `W` aus der Tabelle geliefert.

Memoisierung (2)

- Obwohl die Memoisierung wie die verzögerte Auswertung die Wiederholung der Auswertung mancher Ausdrücke vermeidet, lässt sie sich nicht auf die verzögerte Auswertung zurückführen.
- Die verzögerte Auswertung vermeidet die wiederholten Auswertungen, die sich aus einer Auswertung in normaler Reihenfolge ergeben.
- Die verzögerte Auswertung vermeidet NICHT die wiederholten Auswertungen, die sich aus der Definition der Funktion ergeben.
- Eine verzögerte Auswertung der Fibonacci-Funktion ohne Memoisierung führt zu wiederholten Auswertungen.
- Die meisten Programmiersprachen verwenden keine Memoisierung.

Prozedur versus Prozess

- Die Syntax einer Prozedur (oder eines Programms) bestimmt nicht allein, welche Gestalt der zugehörige (Berechnungs-) Prozess hat.
- Der (Berechnungs-) Prozess hängt sowohl von der Prozedur (oder dem Programm) als auch vom Auswertungsalgorithmus ab.
- Die Auswertungsalgorithmen von modernen funktionalen Programmiersprachen erkennen die Endrekursion und erzeugen aus endrekursiven Programmen iterative Prozesse.

Überblick

1. Die „Prozedur“ als Kernbegriff der Programmierung
2. Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen
3. Prozeduren versus Prozesse
4. Ressourcenbedarf – Größenordnungen
5. Beispiel: Der größte gemeinsame Teiler

Ressourcenbedarf – Größenordnungen

- Prozesse verbrauchen zwei Ressourcen:
  - Rechenzeit
  - Speicherplatz
- Die folgenden Größenordnungen geben an, wie viel Rechenzeit oder Speicherplatz verbraucht werden.

Laufzeitanalyse (1)

- *1. Ansatz*: Direktes **Messen der Laufzeit** (z.B. in ms):
  - Abhängig von vielen Parametern, wie Rechnerkonfiguration, Rechnerlast, Compiler, Betriebssystem, …
  - Deshalb: kaum übertragbar und ungenau
- *2. Ansatz*: Zählen der benötigten **Elementaroperationen** des Algorithmus in Abhängigkeit von der Größe *n* der Eingabe
  - Das algorithmische Verhalten wird als Funktion der benötigten Elementaroperationen dargestellt.
  - Die Charakterisierung dieser elementaren Operationen ist abhängig von der jeweiligen Problemstellung und dem zugrunde liegenden Algorithmus.
  - Beispiele für Elementaroperationen: Zuweisungen, Vergleiche, arithmetische Operationen, Zeigerdereferenzierungen oder Arrayzugriffe

Laufzeitanalyse (2)

- Das Maß für die Größe *n* der Eingabe ist abhängig von der Problemstellung, z.B.
  - Suche eines Elementes in einer Liste: *n* = Anzahl der Elemente
  - Multiplikation zweier Matrizen: *n* = Dimension der Matrizen
  - Sortierung einer Liste von Zahlen: *n* = Anzahl der Zahlen
  - Berechnung der *k*-ten Fibonacci-Zahl: *n* = *k*

- Laufzeit = benötigte Elementaroperationen bei einer bestimmten Eingabelänge *n*

Analog:

- Speicherplatz = benötigter Speicher bei einer bestimmten Eingabelänge *n*

Problem: Hardware-Abhängigkeit

- Laufzeit einzelner Elementar-Operation ist abhängig von der eingesetzten Rechner-Hardware.
- Frühere Rechner (incl. heutige PDAs, Mobiltelefone…):
  - „billig“: Fallunterscheidungen, Wiederholungen
  - „mittel“: Rechnen mit ganzen Zahlen (Multiplikation usw.)
  - „teuer“: Rechnen mit reellen Zahlen
- Heutige Rechner (incl. z.B. Spiele-Konsolen):
  - „billig“: Rechnen mit reellen und ganzen Zahlen
  - „teuer“: Fallunterscheidungen

Asymptotisches Laufzeitverhalten

- „Kleine“ Probleme (z.B. n=5) sind uninteressant:
  Die Laufzeit des Programms ist eher bestimmt durch die Initialisierungskosten (Betriebssystem, Programmiersprache etc.) als durch den Algorithmus selbst.
- Interessanter:
  - Wie verhält sich der Algorithmen bei sehr großen Problemgrößen?
  - Wie verändert sich die Laufzeit, wenn ich die Problemgröße variiere (z.B. Verdopplung der Problemgröße)

➔ asymptotisches Laufzeitverhalten

Definition *O*-Notation

- Mit der *O*-Notation haben Informatiker einen Weg gefunden, die asymptotische Komplexität (bzgl. Laufzeit oder Speicherplatzbedarf) eines Algorithmus zu charakterisieren.
- Definition *O*-Notation:

Seien $f : \mathrm{N} \rightarrow \mathrm{N}$ und $s : \mathrm{N} \rightarrow \mathrm{N}$ zwei Funktionen (s wie Schranke).
Die Funktion f ist von der Größenordnung $O(s)$, geschrieben $f \in O(s)$, wenn es $k \in \mathrm{N}$ und $m \in \mathrm{N}$ gibt, so dass gilt:
Für alle $n \in \mathrm{N}$ mit $n \geq m$ ist $f(n) \leq k * s(n)$.

Bemerkungen zur *O*-Notation

- k ist unabhängig von n:
  k muss dieselbe Konstante sein, die für alle $n \in \mathrm{N}$ garantiert, dass $f(n) \leq k * s(n)$.
- Existiert KEINE solche Konstante k, ist f nicht von der Größenordnung $O(s)$.
- $O(s)$ bezeichnet die Menge aller Funktionen, die bezüglich s die Eigenschaft aus der Definition haben.
- Man findet in der Literatur häufig $f = O(s)$ statt $f \in O(s)$. Da f nicht gleichzeitig Element von $O(s)$ und gleich zu $O(s)$ sein kann, ist das irreführend.

Rechnen mit der *O*-Notation

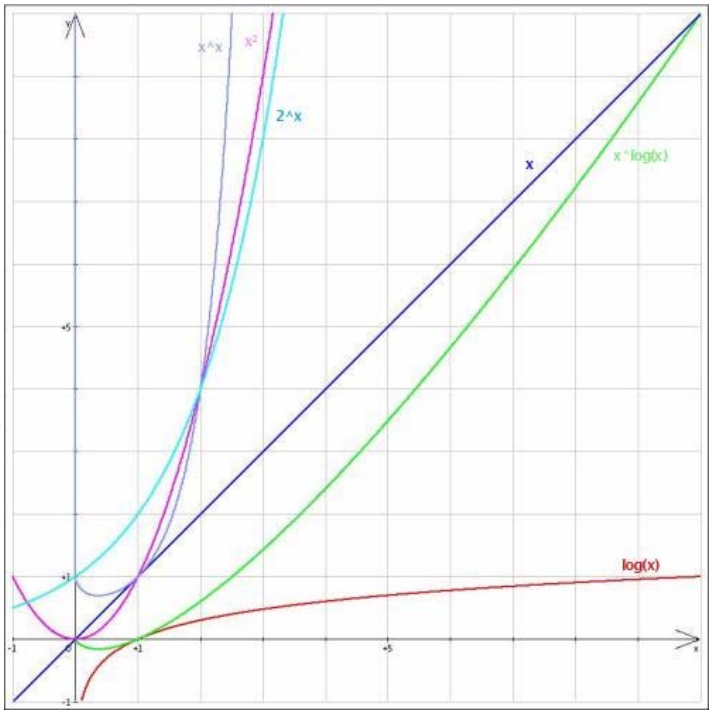
- Elimination von Konstanten:
  - $2 \cdot n \in O(n)$
  - $n/2 + 1 \in O(n)$
- Bei einer Summe zählt nur der am stärksten wachsende Summand (mit dem höchsten Exponenten):
  - $2n^3+5n^2+10n+20 \in O(n^3)$
  - $O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n)$ $\subseteq O(n^2) \subseteq O(n^3) \subseteq \ldots \subseteq O(2^n)$
- Beachte: 1000 n^2 ist nach diesem Leistungsmaß immer „besser“ als 0,001 n^3, auch wenn das m, ab dem die $O(n^2)$-Funktion unter der $O(n^3)$-Funktion verläuft, sehr groß ist (m=1 Million)

Wichtige Klassen von Funktionen (1)

	Sprechweise	**Typische Algorithmen / Operationen**
$O(1)$	konstant	Addition, Vergleichsoperationen, rekursiver Aufruf, …
$O(\log n)$	logarithmisch	Suchen auf einer sortierten Menge
$O(n)$	linear	Bearbeiten jedes Elementes einer Menge
$O(n \cdot \log n)$		gute Sortierverfahren
$O(n \cdot \log^2 n)$		
…		
$O(n^2)$	quadratisch	primitive Sortierverfahren
$O(n^k)$, k≥2	polynomiell	
…		
$O(2^n)$	exponentiell	Ausprobieren von Kombinationen

Wichtige Klassen von Funktionen (2)

- Die *O*-Notation hilft insbesondere bei der Beurteilung, ob ein Algorithmus für großes *n* noch geeignet ist bzw. erlaubt einen Effizienz-Vergleich zwischen verschiedenen Algorithmen für große *n*.
- Schlechtere als polynomielle Laufzeit gilt als nicht effizient.



Beispiel: Laufzeitanalyse Fakultätsfunktion

- `fak(n) = if n = 0 then 1`
  `else n * fak(n - 1);`
- Sei T_{fak} die Laufzeit von `fak`.
- Behauptung: $T_{fak}(n) \in O(n)$
- Es gilt:

$$\begin{aligned} T_{fak}(n) &\leq c + T_{fak}(n-1) \\ &\leq c + (c + T_{fak}(n-2)) \\ &\leq 3c + T_{fak}(n-3) \\ &\leq kc + T_{fak}(n-k) \end{aligned}$$

Beweis mit vollständiger Induktion über *k*

- Damit: $T_{fak}(n) \leq nc + T_{fak}(0) \leq nc + c \in O(n)$

Überblick

1. Die „Prozedur“ als Kernbegriff der Programmierung
2. Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen
3. Prozeduren versus Prozesse
4. Ressourcenbedarf – Größenordnungen
5. Beispiel: Der größte gemeinsame Teiler

Begriffserklärung, Notation

- Der größte gemeinsame Teiler (ggT) zweier natürlicher Zahlen *a* und *b* ist die größte natürliche Zahl, durch die sowohl *a* als auch *b* teilbar (d.h. ohne Rest dividierbar) ist.
- Die Notation $t|a$ wird verwendet, um auszudrücken, dass *t* ein Teiler von *a* ist (d.h. es gibt ein $k \in \mathrm{N}$ mit $a = t * k$).
- Ist *t* gemeinsamer Teiler von *a* und *b*, schreibe $t|a$ und $t|b$.

Primfaktorzerlegung

- Der ggT von zwei natürlichen Zahlen a und b kann leicht aus der Primfaktorzerlegung von a und b ermittelt werden.
- Er ist das Produkt aller p^n mit:
  - die Primzahl p kommt in jeder der beiden Zerlegungen (einmal) vor, einmal mit Exponent n_1, einmal mit Exponent n_2.
  - n ist das Minimum von n_1 und n_2.
- Die Zerlegung einer natürlicher Zahl in Primfaktoren ist eine zeitaufwendige Aufgabe, so dass dieser Ansatz zur Berechnung des ggT zweier natürlicher Zahlen ziemlich ineffizient ist.

Effizienter Ansatz zur Berechnung des ggT zweier natürlicher Zahlen

Satz:
Seien $a \in \mathrm{N}$ und $b \in \mathrm{N}$ mit $a \geq b$.
Sei r der Rest der Ganzzahldivision von a durch b (d.h. $a = (b * c) + r$ für ein $c \in \mathrm{N}$). Sei $t \in \mathrm{N}$.

$t|a$ und $t|b$ genau dann, wenn $t|b$ und $t|r$.

- Daraus folgt, dass der $ggT(a, b)$ zweier natürlicher Zahlen a und b mit $a \geq b$ und $a = b * c + r$ gleich dem $ggT(b, r)$ von b und r ist.

Beweis: Notwendige Bedingung (von links nach rechts „=>“)

- Seien a, b, c und r wie im Satz definiert.
- Sei angenommen, dass $t|a$ und $t|b$.
- Zu zeigen ist, dass $t|b$ und $t|r$ gelten.
- Da nach Annahme $t|b$ gilt, reicht es, $t|r$ zu zeigen.

  Da $t|b$ gilt, gibt es t_b mit $b = t * t_b$.

  Da $t|a$ gilt, gibt es t_a mit $a = t * t_a$.
- Nach Annahme gilt $a = b * c + r$

  also $t * t_a = a = b * c + r = t * t_b * c + r$,

  also $r = t * t_a - t * t_b * c = t * (t_a - t_b * c)$, d.h. $t|r$

Beweis: Hinreichende Bedingung (von rechts nach links „<=“)

- Seien a, b, c und r wie im Satz definiert.
- Sei angenommen, dass $t|b$ und $t|r$.
- Zu zeigen ist, dass $t|a$ und $t|b$ gelten.
- Da nach Annahme $t|b$ gilt, reicht es, $t|a$ zu zeigen.

  Da $t|b$ gilt, gibt es t_b mit $b = t * t_b$.

  Da $t|r$ gilt, gibt es t_r mit $r = t * t_r$.
- Nach Annahme gilt $a = b * c + r$

  also $a = t * t_b * c + t * t_r = t * (t_b * c + t_r)$, d.h. $t|a$

qed.

Rekursiver Ansatz zur Berechnung des ggT

1. Wenn $a < b$, dann vertausche a und b.
2. Andernfalls (d.h. $a \geq b$) ist der ggT von a und b der ggT von b und r, wobei r der Rest der Ganzzahldivision von a durch b ist (d.h. in SML $r = a$ mod b).

- Die aktuellen Parameter (a, b) werden paarweise bei jedem rekursiven Aufruf (außer beim ersten, falls $a < b$ ist) echt kleiner, da der Rest einer Ganzzahldivision durch b echt kleiner als b ist.
- Der Basisfall ist $ggT(a, 0) = a$, da $0 = a * 0 + 0$.

Terminierung (Informeller Beweis)

- Bei jedem rekursiven Aufruf ist der zweite aktuelle Parameter (d.h. der Wert des formalen Parameters b) eine natürliche Zahl, die echt kleiner als der zweite aktuelle Parameter des vorherigen rekursiven Aufrufs ist.
- Zudem ist diese natürliche Zahl größer gleich 0.
- Da es zwischen dem Wert des formalen Parameters b beim ersten rekursiven Aufruf und 0 nur endlich viele natürliche Zahlen gibt, muss nach endlich vielen rekursiven Aufrufen der formale Parameter b den Wert 0 haben.

qed.

Endrekursive Funktion zur Berechnung des ggT

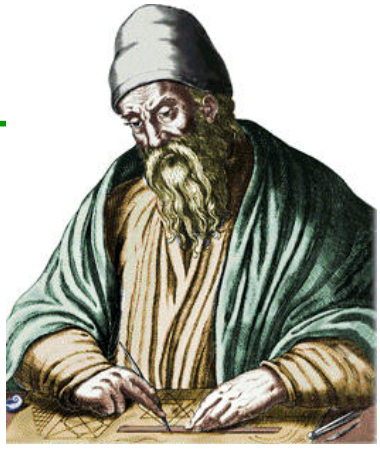
Die Auswertung dieser Funktion löst einen iterativen (Berechnungs-) Prozess aus.

```
fun ggT(a,b) =
   if a<b then ggT(b,a)
          else if b = 0 then a
                        else ggT(b,a mod b);
```

- Der Algorithmus, den die Funktion `ggT` implementiert, konvergiert sehr schnell:

```
ggT(150, 60)
ggT(60, 30)
ggT(30, 0)
30
```


Euklid



- Dieser Algorithmus zur Berechnung des ggT zweier natürlicher Zahlen wird Euklid (ca. 3. Jh. vor Christus) zugeschrieben, weil er in Euklids „Elemente der Mathematik“ erwähnt ist.
- Er gilt als der älteste bekannte Algorithmus, weil er im Gegensatz zu anderen überlieferten Algorithmen aus älteren oder sogar jüngeren Zeiten nicht mittels Beispielen, sondern abstrakt (mit Redewendungen anstelle von Variablen) spezifiziert ist.

Abschätzung der Rechenzeit des Euklid´schen Algorithmus

Satz (Lamé)
Seien $a \in \mathbb{N}$ und $b \in \mathbb{N}$, so dass $a \geq b$ ist.
Benötigt der Euklid´sche Algorithmus zur Berechnung von $ggT(a, b)$ insgesamt n Rekursionsschritte, so gilt $b \geq fib(n)$, wobei $fib(n)$ die n-te Fibonacci-Zahl ist.

Beweis des Satzes (1)

- Betrachte:

  $ggT(a_0, b_0)$
  $ggT(a_1, b_1)$
  $ggT(a_2, b_2)$

- Nach Definition gilt:

  $a_1 = b_0$
  $a_2 = b_1$
  $b_1 = a_0 \bmod b$ d.h. $a_0 = b_0 * c_0 + b_1$ (für ein $c_0 \in \mathbb{N}$)
  $b_2 = a_1 \bmod b_1$ d.h. $a_1 = b_1 * c_1 + b_2$ (für ein $c_1 \in \mathbb{N}$)
  Da $c_0 \geq 1$ und $c_1 \geq 1$ ist, folgt:
  (*) $a_0 \geq b_0 + b_1$
  $a_1 = b_0 \geq b_1 + b_2$

Beweis des Satzes (2)

- Dieses Ergebnis entspricht dem Bildungsgesetz für die Fibonacci-Zahlen.
- Deshalb kann man die Hypothese aufstellen, dass

  $\texttt{b}_0 \geq$ *fib*(n), wenn n die Anzahl der rekursiven Aufrufe der Funktion `ggT` während der Berechnung von $(\texttt{a}_0, \texttt{b}_0)$ ist (wobei dieser allererste Aufruf nicht mitgezählt wird).

Beweis mit vollständiger Induktion

- Verschärfe die Behauptung:

  Für alle $n \in$ N gilt: für alle $k \leq n$ und für alle $\texttt{a} \in$ N und $\texttt{b} \in$ N mit $\texttt{a} \geq \texttt{b}$, für die die Auswertung von `ggT(a,b)` genau k rekursive Aufrufe benötigt, gilt $\texttt{b} \geq$ *fib*(k).
- Daraus folgt die eigentliche Hypothese als Spezialfall mit $k = n$.
- Vollständige Induktion:
  - Basisfall / Basisfälle
  - Induktionsannahme
  - Induktionsfall

Basisfälle

- $n = 0$

  für alle $k \leq n$ und für alle `a` $\in$ N und `b` $\in$ N mit `a` $\geq$ `b`, für die die Auswertung von `ggT(a,b)` genau k rekursive Aufrufe benötigt, gilt `b` $\geq$ $fib(k)$, da k = 0 und $fib(k) = 0$ und `b` $\in$ N gilt.

- $n = 1$

  Für $k = 1$ seien `a` $\in$ N und `b` $\in$ N mit `a` $\geq$ `b`, so dass die Auswertung von `ggT(a,b)` genau k rekursive Aufrufe benötigt, also genau 1. Das bedeutet, dass `a` und `b` die Bedingungen erfüllen müssen, die in der Definition von `ggT(a,b)` zum letzten else-Fall führen, also insbesondere b $\neq$ 0.

  Also gilt b $\geq 1 = fib(1) = fib(k)$.

  Also ist die Behauptung für alle $k \leq 1$ gezeigt.

Induktionsannahme

- Für alle $k \leq n$ und für alle `a` $\in$ N und `b` $\in$ N mit `a` $\geq$ `b`, für die die Auswertung von `ggT(a,b)` genau k rekursive Aufrufe benötigt, gilt `b` $\geq$ $fib(k)$.

Induktionsfall (1)

- Sei $n \geq 2$ und es gelte die Induktionsannahme.
- Sei nun $k \leq n + 1$ und seien $\texttt{a} \in \mathbb{N}$ und $\texttt{b} \in \mathbb{N}$ mit $\texttt{a} \geq \texttt{b}$, so dass die Auswertung von `ggT(a,b)` genau k rekursive Aufrufe benötigt. Falls $k \leq n$ ist, gilt $\texttt{b} \geq fib(k)$ nach Induktionsannahme.

  Es bleibt also nur noch der Fall $k = n + 1$ zu zeigen.

Induktionsfall (2)

- Die Auswertung von `ggT(a,b)` benötige also $n + 1$ rekursive Aufrufe. Sei $\texttt{a}_0 = \texttt{a}$ und $\texttt{b}_0 = \texttt{b}$ und seien $\texttt{ggT}(\texttt{a}_1,\texttt{b}_1)$ und $\texttt{ggT}(\texttt{a}_2,\texttt{b}_2)$ die zwei ersten rekursiven Aufrufe.

  Nach Konstruktion benötigt die Auswertung von $\texttt{ggT}(\texttt{a}_1,\texttt{b}_1)$ genau n rekursive Aufrufe und die Auswertung von $\texttt{ggT}(\texttt{a}_2,\texttt{b}_2)$ genau $n - 1$ rekursive Aufrufe.

  Es gilt sowohl $n \leq n$ als auch $n - 1 \leq n$, so dass nach Induktionsannahme gilt $\texttt{b}_1 \geq fib(n)$ und $\texttt{b}_2 \geq fib(n - 1)$.

  Nach `(*)` gilt: $\texttt{b}_0 \geq \texttt{b}_1 + \texttt{b}_2$

  Also $\texttt{b} = \texttt{b}_0 \geq fib(n) + fib(n - 1) = fib(n + 1) = fib(k)$

  qed.

Bezug zum Goldenen Schnitt

- Benötigt der Euklid´sche Algorithmus zur Berechnung von `ggT(a,b)` genau *n* rekursive Aufrufe, so gilt nach dem Satz von Lamé:

  $b \geq fib(n) \approx \Phi^n$ (Φ ist der Goldene Schnitt)

- Daraus folgt, dass asymptotisch, also für große *n*, gilt (mit der Schreibweise $\log_\Phi$ für den Logarithmus zur Basis Φ):

  $\log_\Phi b \geq n$

- Da es aber eine Konstante *k* gibt mit $\log_\Phi b \leq k * \ln b$, wobei `ln` den Logarithmus zur Basis *e* bezeichnet, gilt asymptotisch auch $n \leq k * \ln b$, also

  $ggT(a,b) \in O(\ln(b))$