1. Rekursion

Rekursive Funktionen, Korrektheit, Terminierung,
Rekursion vs. Iteration

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar, das heisst
- die Funktion erscheint in ihrer eigenen Definition.

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar, das heisst
- die Funktion erscheint in ihrer eigenen Definition.

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{falls } n > 1 \end{cases}$$

Rekursion in C++

- modelliert oft direkt die mathematische Rekursionsformel.

Rekursion in C++

- modelliert oft direkt die mathematische Rekursionsformel.

```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Unendliche Rekursion

- ist so leicht zu erzeugen wie eine unendliche Schleife,
- sollte aber genauso vermieden werden.

Unendliche Rekursion

- ist so leicht zu erzeugen wie eine unendliche Schleife,
- sollte aber genauso vermieden werden.

```
void f()
{
  f(); // calls f(), calls f(), calls f(),...
}
```

Terminierung von rekursiven Funktionsaufrufen

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

Terminierung von rekursiven Funktionsaufrufen

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

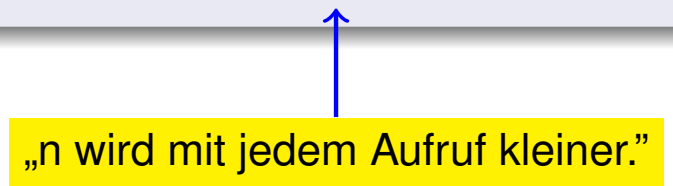
`fac(n)`:
terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument < n aufgerufen.

Terminierung von rekursiven Funktionsaufrufen

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fac(n)`:
terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument < n aufgerufen.



Auswertung rekursiver Funktionsaufrufe (z.B. fac(4))

```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Aufruf von `fac(4)`

Auswertung rekursiver Funktionsaufrufe (z.B. fac(4))

```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{ // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments mit dem Wert des Aufrufarguments

Auswertung rekursiver Funktionsaufrufe (z.B. fac(4))

```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{ // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Ausführen des Funktionsrumpfs: Auswertung des Rückgabeausdrucks

Auswertung rekursiver Funktionsaufrufe (z.B. fac(4))

```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{ // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Ausführen des Funktionsrumpfs: Rekursiver Aufruf von fac mit Aufrufargument $n - 1 == 3$

Auswertung rekursiver Funktionsaufrufe (z.B. fac(4))

```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments mit dem Wert des Aufrufarguments

Auswertung rekursiver Funktionsaufrufe (z.B. fac(4))

```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Es gibt jetzt zwei n. Das von `fac(4)` und das von `fac(3)`

Initialisierung des formalen Arguments mit dem Wert des Aufrufarguments

Auswertung rekursiver Funktionsaufrufe (z.B. fac(4))

```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Es wird mit dem n des aktuellen Aufrufs fortgefahren: $n = 3$

Initialisierung des formalen Arguments mit dem Wert des Aufrufarguments

Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

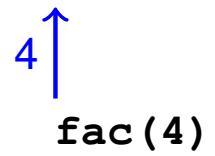
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht

```
fac(4)
```

Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

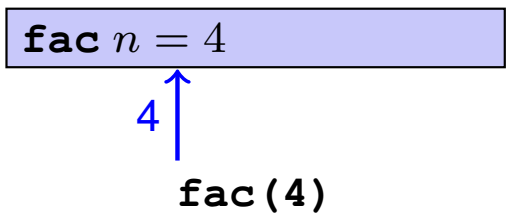
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

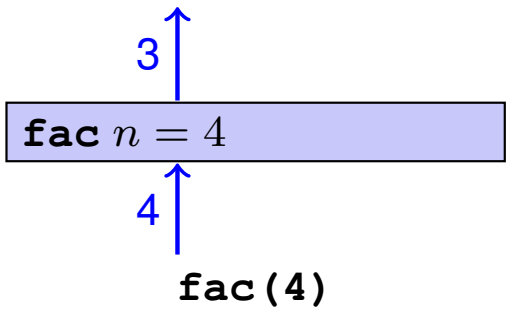
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

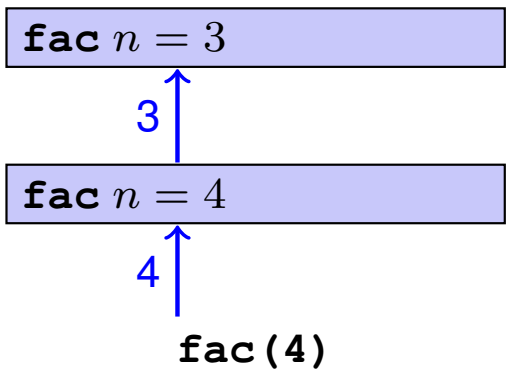
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

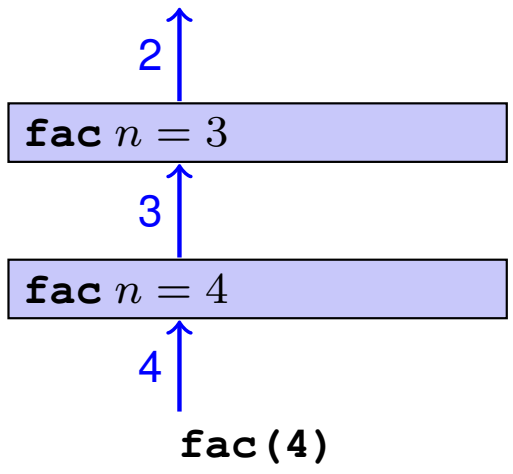
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

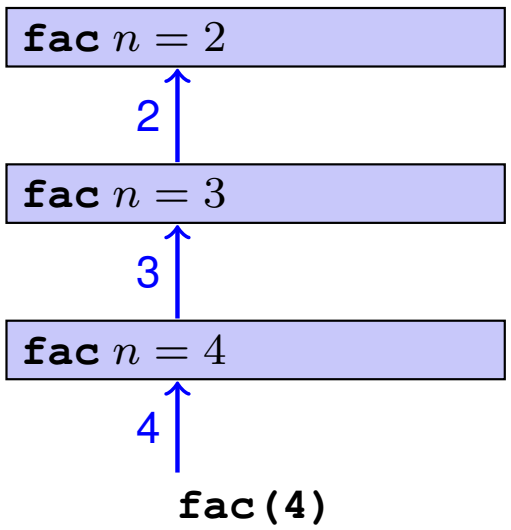
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

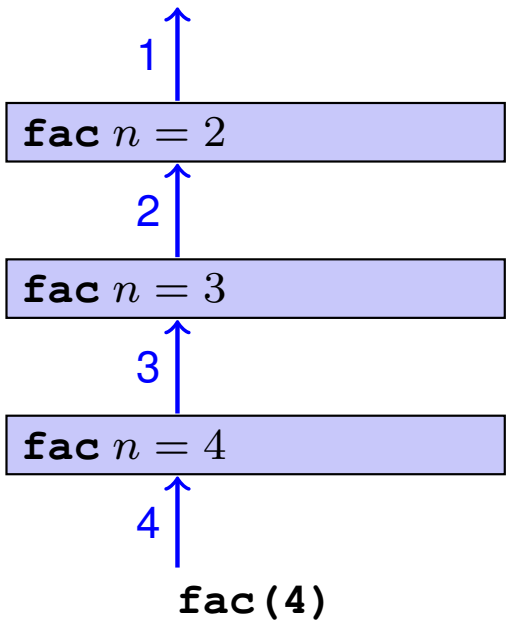
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

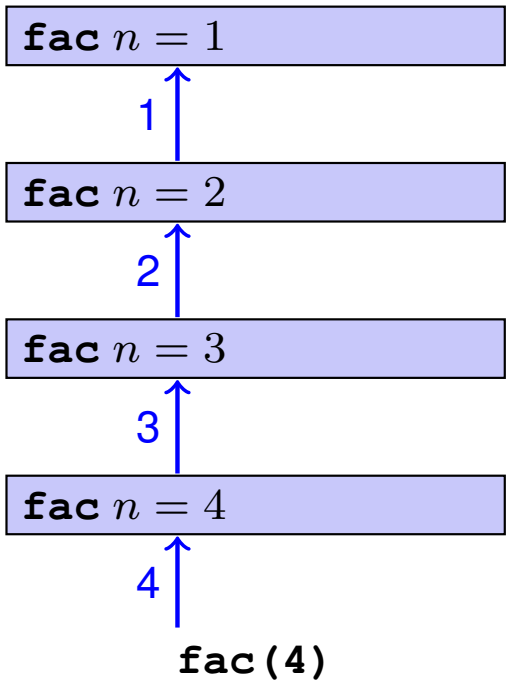
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

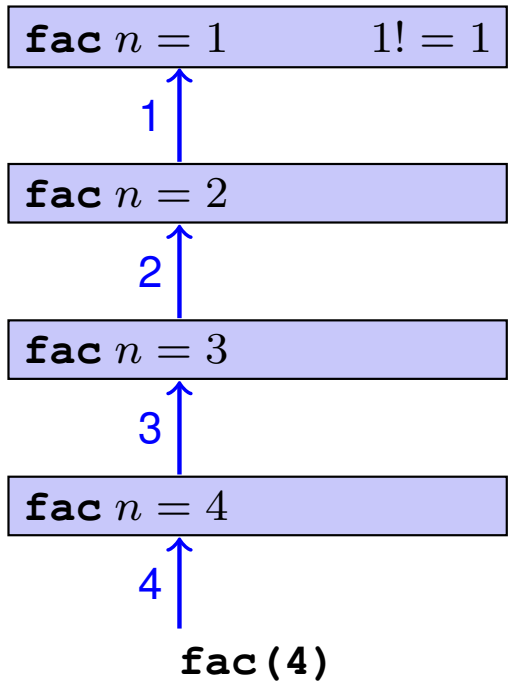
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

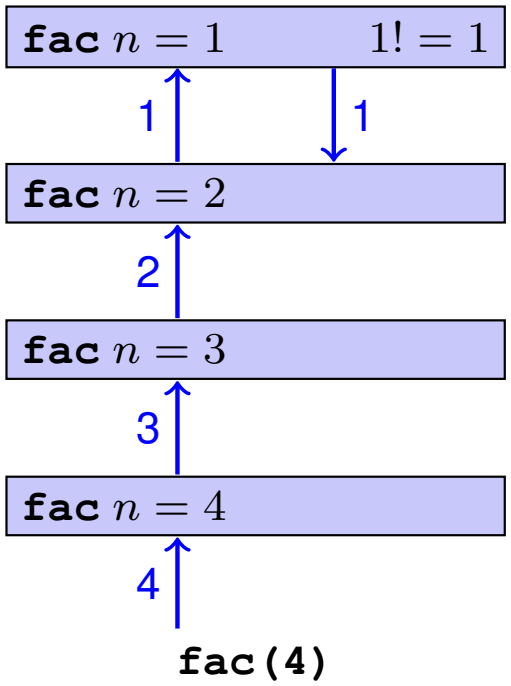
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

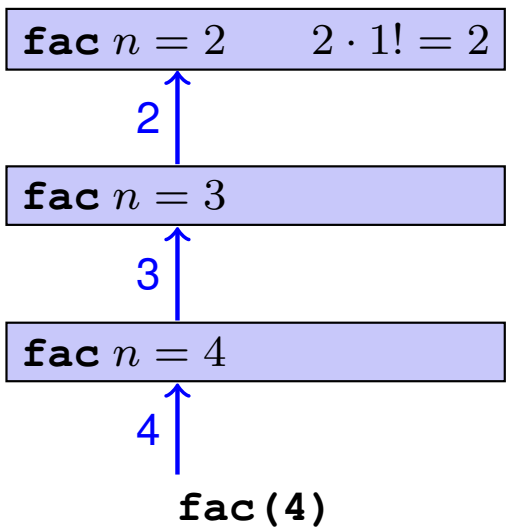
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

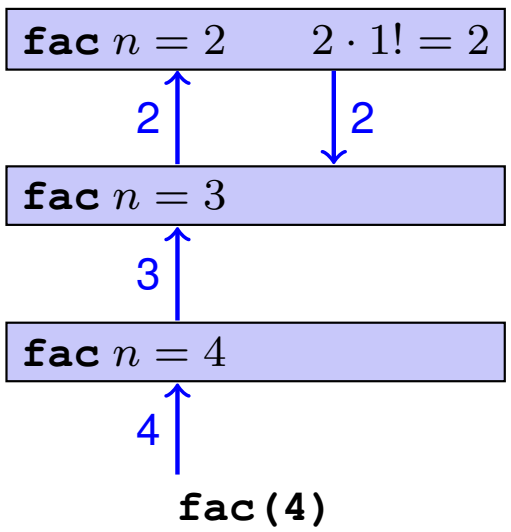
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

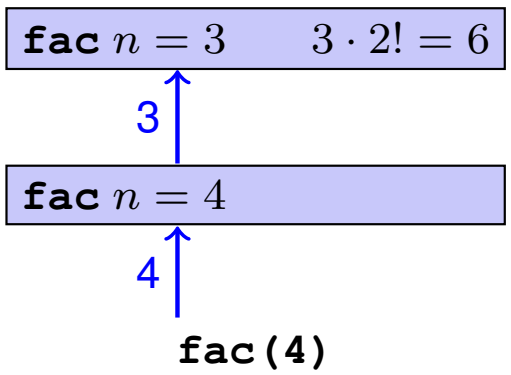
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

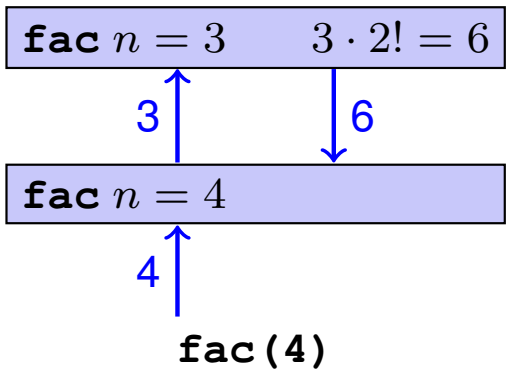
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

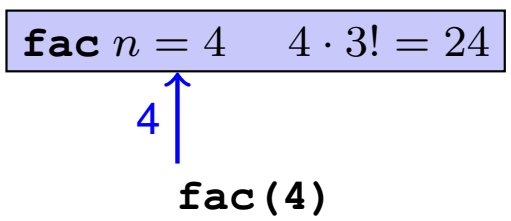
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

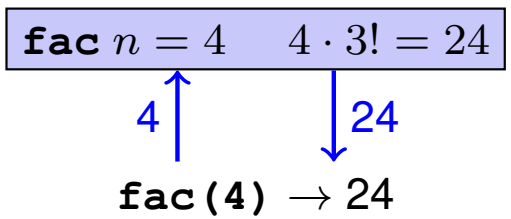
- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht



Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

- Wert des Aufrufarguments kommt auf einen Stapel (erst n = 4, dann n = 3,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht

`fac(4)` $\rightarrow$ 24

Grösster gemeinsamer Teiler

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\gcd(a, b)$ zweier natürlicher Zahlen a und b

Grösster gemeinsamer Teiler

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\gcd(a, b)$ zweier natürlicher Zahlen a und b
- basiert auf folgendem Lemma (Beweis im Skript):

$\gcd(a, b) = \gcd(b, a \bmod b)$ für $b > 0$.

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd
    (const unsigned int a, const unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd
    (const unsigned int a, const unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Korrektheit

$$\gcd(a, 0) = a$$
$$\gcd(a, b) = \gcd(b, a \bmod b),\ b > 0.$$

↑ Lemma

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd
    (const unsigned int a, const unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Terminierung $a \bmod b < b$, also wird b in jedem rekursiven Aufruf kleiner

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd
    (const unsigned int a, const unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Korrektheit und Terminierung müssen stets separat bewiesen werden!

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd
    (const unsigned int a, const unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Beispiel

`return gcd (b, a); // b != 0` wäre auch korrekt ($\gcd$ ist kommutativ), terminiert aber nicht.

Fibonacci Zahlen

$$F_n := \begin{cases} 0 & , \text{falls } n = 0 \\ 1 & , \text{falls } n = 1 \\ F_{n-1} + F_{n-2} & , \text{falls } n > 1 \end{cases}$$

Fibonacci Zahlen

$$F_n := \begin{cases} 0 & , \text{falls } n = 0 \\ 1 & , \text{falls } n = 1 \\ F_{n-1} + F_{n-2} & , \text{falls } n > 1 \end{cases}$$

$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Fibonacci Zahlen

Fibonacci Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (const unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Fibonacci Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (const unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Korrrektheit und Terminierung sind klar.

Fibonacci Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (const unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Laufzeit

`fib(50)` dauert „ewig”, denn es berechnet F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal, F_{45} 8-mal, F_{44} 13-mal, F_{43} 21-mal ... F_1 ca. 10^9 mal (!)

Rekursion und Iteration

Rekursion kann im Prinzip ersetzt werden durch

- Iteration (Schleifen)
- expliziten „Aufrufstapel” (z.B. Feld).

Oft sind direkte rekursive Formulierungen einfacher, aber manchmal auch weniger effizient.

Endrekursion

Eine Funktion ist endrekursiv, wenn sie genau einen rekursiven Aufruf ganz am Ende enthält.

Endrekursion

Eine Funktion ist endrekursiv, wenn sie genau einen rekursiven Aufruf ganz am Ende enthält.

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd
    (const unsigned int a, const unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Endrekursion

ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

Endrekursion

ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

$$(a, b) \longrightarrow (b, a \bmod b)$$

Endrekursion

ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

$$(a, b) \longrightarrow (b, a \bmod b)$$

Endrekursion

ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

$$(a, b) \longrightarrow (b, a \bmod b)$$

Endrekursion

ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

$$(a, b) \longrightarrow (b, a \bmod b)$$

Aber die rekursive Version ist lesbarer und (fast) genauso effizient.

Fibonacci-Zahlen iterativ

Idee

- berechne jede Zahl genau einmal, in der Reihenfolge F_0 , F_1 , F_2 , F_3 , ...
- speichere die jeweils letzten beiden berechneten Zahlen (Variablen **`a`**, **`b`**), dann kann die nächste Zahl durch eine Addition erhalten werden.

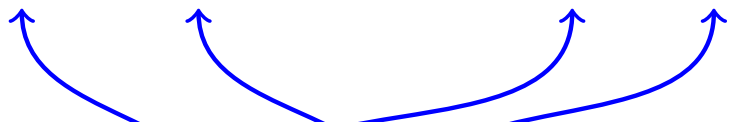
Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (const unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i)
  {
    unsigned int a_prev = a;  // F_i-2
    a = b;                    // F_i-1
    b += a_prev;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

Fibonacci-Zahlen iterativ

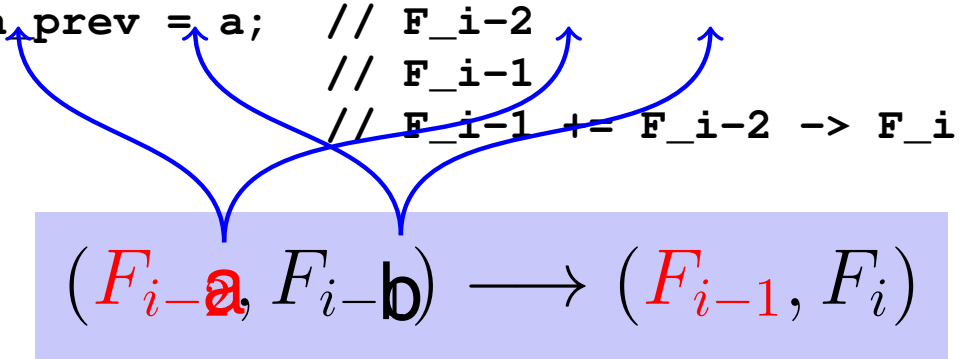
```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (const unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i)
  {
    unsigned int a_prev = a;  // F_i-2
    a = b;                    // F_i-1
    b += a_prev;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$



Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (const unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i)
  {
    unsigned int a_prev = a;  // F_i-2
    a = b;                    // F_i-1
    b += a_prev;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```



Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (const unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i)
  {
    unsigned int a_prev = a;  // F_i-2
    a = b;                    // F_i-1
    b += a_prev;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (const unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i)
  {
    unsigned int a_prev = a;  // F_i-2
    a = b;                    // F_i-1
    b += a_prev;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

sehr schnell auch bei **`fib2(50)`**

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

Die Ackermann-Funktion

$$A(m,n) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0, n = 0 \\ A(m-1, A(m,n-1)), & m > 0, n > 0 \end{cases}$$

Die Ackermann-Funktion

$$A(m,n) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & m > 0, n = 0 \\ A(m-1, A(m,n-1)), & m > 0, n > 0 \end{cases}$$

- ist berechenbar, aber nicht primitiv rekursiv (man dachte Anfang des 20. Jahrhunderts, dass es diese Kombination gar nicht gibt)
- wächst extrem schnell

Die Ackermann-Funktion

```
// POST: return value is the Ackermann
//       function value A(m,n)
unsigned int A (const unsigned int m,
                const unsigned int n)
{
  if (m == 0) return n+1;
  if (n == 0) return A(m-1,1);
  return A(m-1, A(m, n-1));
}
```

Die Ackermann-Funktion

n →

m ↓

	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1						
2						
3						
4						

Die Ackermann-Funktion

n →

m ↓

	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2						
3						
4						

Die Ackermann-Funktion

n →

m ↓

	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3						
4						

Die Ackermann-Funktion

n →

m ↓

	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3	5	13	29	61	...	$2^{n+3}-3$
4						

Die Ackermann-Funktion

n →

m ↓

	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3	5	13	29	61	...	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	...	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}-3$

Die Ackermann-Funktion

n →

m ↓

	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3	5	13	29	61	...	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	...	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}-3$

Turm von $n+3$ Zweierpotenzen !

Die Ackermann-Funktion

n →

m ↓

	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3	5	13	29	61	...	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	...	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}-3$

nicht mehr praktisch berechenbar

Die Ackermann-Funktion – Moral

- Rekursion ist sehr mächtig...
- ... aber auch gefährlich:

Die Ackermann-Funktion – Moral

- Rekursion ist sehr mächtig...
- ... aber auch gefährlich:

Es ist leicht, harmlos aussehende rekursive Funktionen hinzuschreiben, die theoretisch korrekt sind und terminieren, praktisch aber jeden Berechenbarkeitsrahmen sprengen.