Einführung in die Programmierung

Vorlesungsteil 6
Rekursion

PD Dr. Thomas Hinze

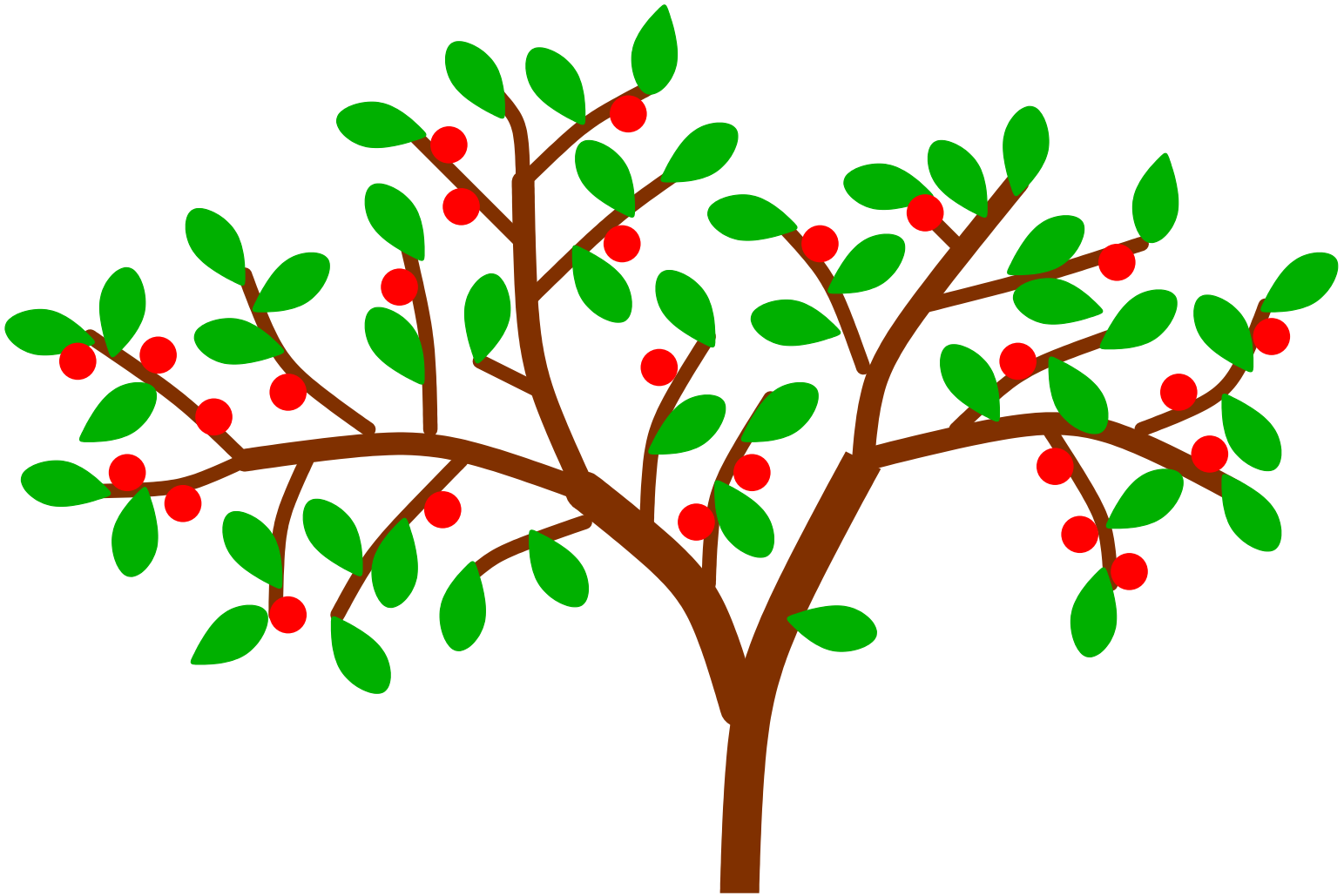
Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik, Informations- und Medientechnik

Wintersemester 2015/2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

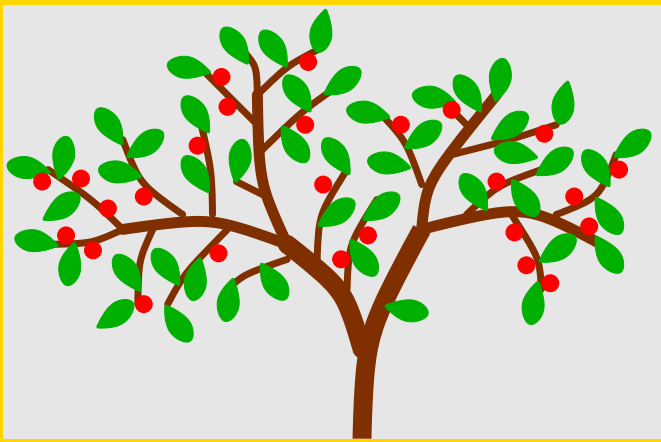
Wieviele Laubblätter und Früchte hängen am Baum?

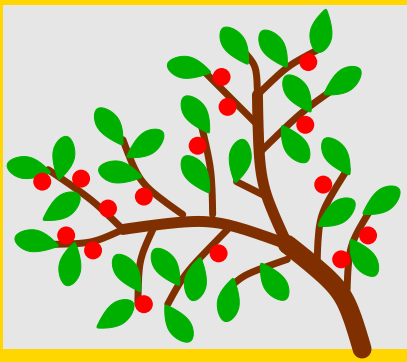
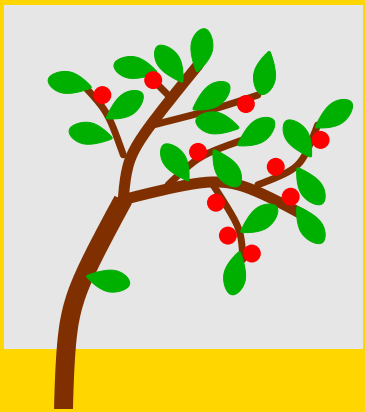
Ein schwieriges, unübersichtliches Problem, wenn man nichts abpflücken darf . . .



Idee: Das Problem in kleinere Probleme zerlegen

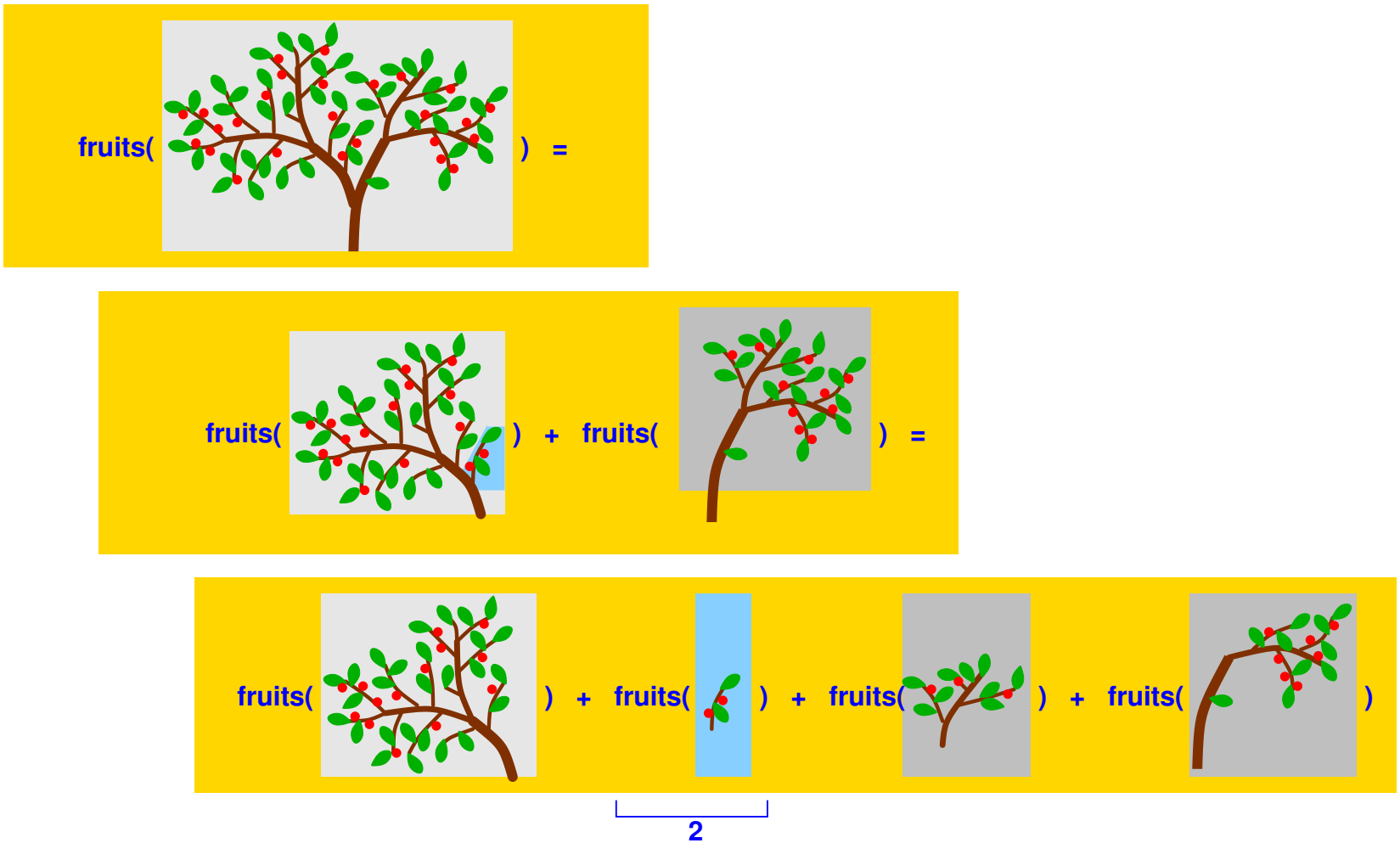
Den Baum gedanklich vom Stamm aus an der ersten Astgabelung teilen

fruits( **) =**

fruits( **) + fruits(**  **)**

Immer weiter zerlegen bis hin zu trivialen Problemen

Ergebnis trivialer Probleme leicht sichtbar



Aus Ergebnissen der Trivialprobleme das Gesamtergebnis zusammensetzen

„Recurrere“ (lat.) heißt „zurückführen“

Manche Aufgaben der Informatik lassen sich gut dadurch lösen, dass man ein *großes Problem* schrittweise auf immer kleinere gleichartige Probleme *zurückführt*, bis *einfache (triviale) Lösungen* entstehen (*rekursiver Abstieg*). Daraus wird schließlich die Lösung des ursprünglichen großen Problems zusammengesetzt (*rekursiver Aufstieg*).

Vorlesung Einführung in die Programmierung mit C

1. **Einführung und erste Schritte** . . . . . Installation C-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . C als Taschenrechner nutzen, Tastatureingabe $\rightarrow$ Formelberechnung $\rightarrow$ Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen und Schleifen programmieren
4. **Aussagenlogik in C** . . . . . Schaltbelegungstabellen aufstellen, optimieren und implementieren
5. **Funktionen selbst programmieren** . . . . . Funktionen als wiederverwendbare Werkzeuge, Werteübernahme und -rückgabe
6. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
7. **Felder und Strukturierung von Daten** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Datensätzen
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeiger, Zeichenketten und Dateiarbeit** . . . . . Texte analysieren, ver- und entschlüsseln, Dateien lesen und schreiben
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Elegante induktive Definition mathem. Funktionen

auf natürlichen Zahlen

Addition ($\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0 + x &= x \\ (n+1) + x &= (n+x) + 1 \end{aligned}$$

Induktiv: Definition *erzeugt* beweisbar korrekte Berechnungsvorschrift (Prinzip der vollständigen Induktion)

Elegante induktive Definition mathem. Funktionen

auf natürlichen Zahlen

Addition ($\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0 + x &= x \\ (n+1) + x &= (n + x) + 1 \end{aligned}$$

Multiplikation ($\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0 * x &= 0 \\ (n+1) * x &= n * x + x \end{aligned}$$

Induktiv: Definition *erzeugt* beweisbar korrekte Berechnungsvorschrift (Prinzip der vollständigen Induktion)

Elegante induktive Definition mathem. Funktionen

auf natürlichen Zahlen

Addition ($\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0 + x &= x \\ (n+1) + x &= (n+x) + 1 \end{aligned}$$

Multiplikation ($\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0 * x &= 0 \\ (n+1) * x &= n * x + x \end{aligned}$$

Potenz ($\mathbb{N}_+ \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x * x^n \end{aligned}$$

Induktiv: Definition *erzeugt* beweisbar korrekte Berechnungsvorschrift (Prinzip der vollständigen Induktion)

Elegante induktive Definition mathem. Funktionen

auf natürlichen Zahlen

Addition ($\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0 + x &= x \\ (n+1) + x &= (n+x) + 1 \end{aligned}$$

Fakultät ($\mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) * n! \end{aligned}$$

Multiplikation ($\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0 * x &= 0 \\ (n+1) * x &= n * x + x \end{aligned}$$

Potenz ($\mathbb{N}_+ \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x * x^n \end{aligned}$$

Induktiv: Definition *erzeugt* beweisbar korrekte Berechnungsvorschrift (Prinzip der vollständigen Induktion)

Elegante induktive Definition mathem. Funktionen

auf natürlichen Zahlen

Addition ($\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0 + x &= x \\ (n+1) + x &= (n+x) + 1 \end{aligned}$$

Fakultät ($\mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) * n! \end{aligned}$$

Multiplikation ($\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} 0 * x &= 0 \\ (n+1) * x &= n * x + x \end{aligned}$$

Summe ($\mathbb{N}_+ \longrightarrow \mathbb{N}$)

$$\begin{aligned} \sum_{i=1}^{1} i &= 1 \\ \sum_{i=1}^{n+1} i &= \sum_{i=1}^{n} i + n + 1 \end{aligned}$$

Potenz ($\mathbb{N}_+ \times \mathbb{N} \longrightarrow \mathbb{N}$)

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x * x^n \end{aligned}$$

Induktiv: Definition *erzeugt* beweisbar korrekte Berechnungsvorschrift (Prinzip der vollständigen Induktion)

Rekursion ist in der Programmierung nützlich, weil . . .

- . . . sich zahlreiche numerische Berechnungsvorschriften mathematischer Funktionen elegant, kompakt, eng angelehnt an die induktive mathematische Definition und damit beweisbar korrekt beschreiben und implementieren lassen.

Rekursion ist in der Programmierung nützlich, weil . . .

- . . . sich zahlreiche numerische Berechnungsvorschriften mathematischer Funktionen elegant, kompakt, eng angelehnt an die induktive mathematische Definition und damit beweisbar korrekt beschreiben und implementieren lassen.
- . . . sie in der Regel zu kurzen und dennoch übersichtlichen und gut nachvollziehbaren Quelltexten führt.

Rekursion ist in der Programmierung nützlich, weil . . .

- . . . sich zahlreiche numerische Berechnungsvorschriften mathematischer Funktionen elegant, kompakt, eng angelehnt an die induktive mathematische Definition und damit beweisbar korrekt beschreiben und implementieren lassen.
- . . . sie in der Regel zu kurzen und dennoch übersichtlichen und gut nachvollziehbaren Quelltexten führt.
- . . . sich sehr komplizierte Funktionsverläufe, z.B. äußerst schnell wachsende Funktionen für Benchmark-, Hardware- oder Compilertests, leicht beschreiben lassen, die imperativ (durch Schleifen) nur recht umständlich und schwer verständlich notierbar sind.

Rekursion ist in der Programmierung nützlich, weil . . .

- . . . sich zahlreiche numerische Berechnungsvorschriften mathematischer Funktionen elegant, kompakt, eng angelehnt an die induktive mathematische Definition und damit beweisbar korrekt beschreiben und implementieren lassen.
- . . . sie in der Regel zu kurzen und dennoch übersichtlichen und gut nachvollziehbaren Quelltexten führt.
- . . . sich sehr komplizierte Funktionsverläufe, z.B. äußerst schnell wachsende Funktionen für Benchmark-, Hardware- oder Compilertests, leicht beschreiben lassen, die imperativ (durch Schleifen) nur recht umständlich und schwer verständlich notierbar sind.
- . . . sie den Zugang für bewährte, breit anwendbare und effiziente Problemlösungsstrategien eröffnet wie „dynamische Programmierung“ oder „Teile und herrsche“.

Rekursion ist in der Programmierung nützlich, weil . . .

- . . . sich zahlreiche numerische Berechnungsvorschriften mathematischer Funktionen elegant, kompakt, eng angelehnt an die induktive mathematische Definition und damit beweisbar korrekt beschreiben und implementieren lassen.
- . . . sie in der Regel zu kurzen und dennoch übersichtlichen und gut nachvollziehbaren Quelltexten führt.
- . . . sich sehr komplizierte Funktionsverläufe, z.B. äußerst schnell wachsende Funktionen für Benchmark-, Hardware- oder Compilertests, leicht beschreiben lassen, die imperativ (durch Schleifen) nur recht umständlich und schwer verständlich notierbar sind.
- . . . sie den Zugang für bewährte, breit anwendbare und effiziente Problemlösungsstrategien eröffnet wie „dynamische Programmierung“ oder „Teile und herrsche“.
- . . . sie es auf elegante Weise gestattet, komplizierte große Datenstrukturen wie z.B. Bäume, systematisch zu durchlaufen, um z.B. in Datenbanksystemen effizient nach Datensätzen zu suchen oder Daten aus solchen umfangreichen Strukturen auszuwerten.

Rekursion hat aber auch Schattenseiten . . .

- Programme, die Rekursion verwenden, sind im Allgemeinen recht ressourcenhungrig, brauchen also im Vergleich zu nichtrekursiven Implementierungen häufig deutlich mehr Speicherplatz und nicht selten mehr Zeit.

Rekursion hat aber auch Schattenseiten . . .

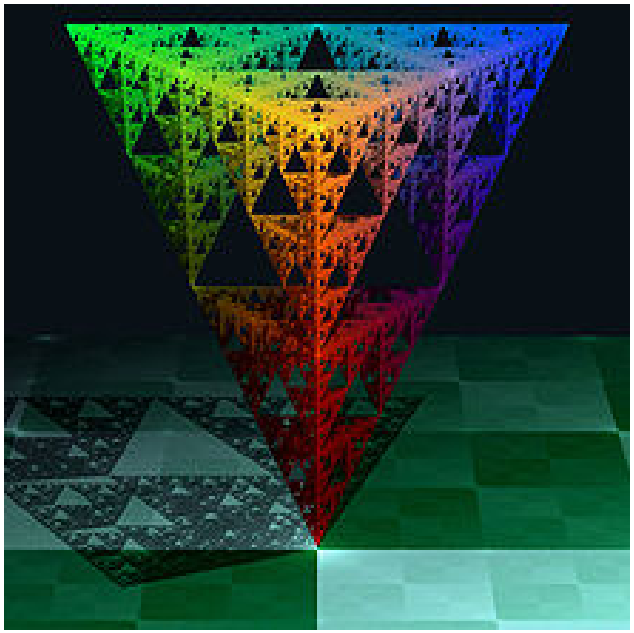
- Programme, die Rekursion verwenden, sind im Allgemeinen recht ressourcenhungrig, brauchen also im Vergleich zu nichtrekursiven Implementierungen häufig deutlich mehr Speicherplatz und nicht selten mehr Zeit.
- Um Rekursion technisch umsetzen zu können, werden höhere Anforderungen an den Compiler und das Betriebssystem gestellt. Beispielsweise muss ein spezieller Speicherbereich, der sogenannte Funktionsaufrufstack, eingerichtet und verwaltet werden.

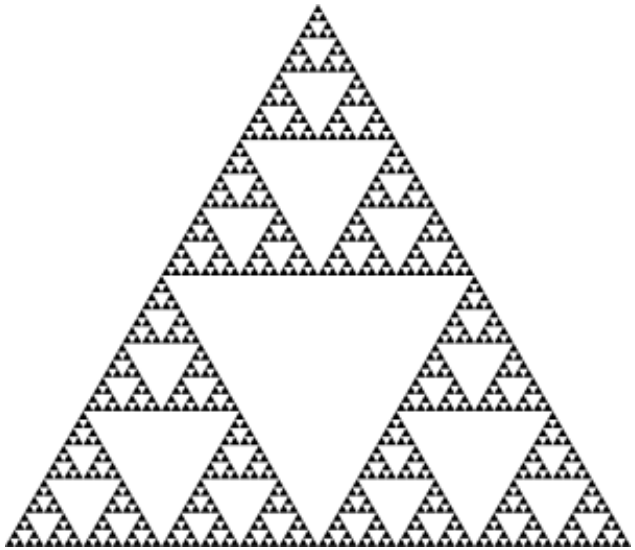
Rekursion hat aber auch Schattenseiten . . .

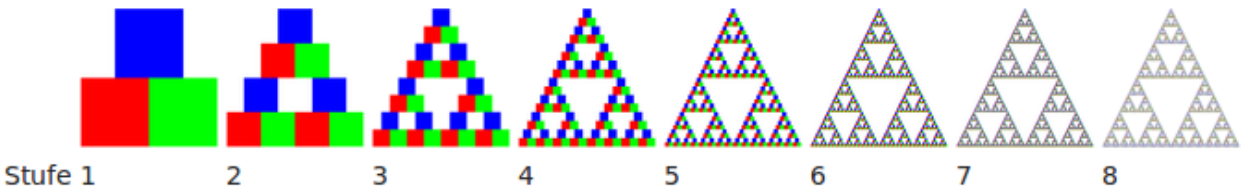
- Programme, die Rekursion verwenden, sind im Allgemeinen recht ressourcenhungrig, brauchen also im Vergleich zu nichtrekursiven Implementierungen häufig deutlich mehr Speicherplatz und nicht selten mehr Zeit.
- Um Rekursion technisch umsetzen zu können, werden höhere Anforderungen an den Compiler und das Betriebssystem gestellt. Beispielsweise muss ein spezieller Speicherbereich, der sogenannte Funktionsaufrufstack, eingerichtet und verwaltet werden.

$\Longrightarrow$ Bei numerischen Berechnungen lohnt es sich, rekursiv programmierte Funktionen durch semantisch äquivalente nichtrekursive Pendants zu ersetzen, sobald der gewünschte Funktionsverlauf feststeht und daran keine Änderungen mehr vorgenommen werden müssen.

Rekursion in der digitalen Kunst







http://de.wikipedia.org/wiki/Sierpinski-Dreieck

Rekursion sorgt für selbstähnliche, immer kleiner werdende Strukturen, die sich zu geometrischen Mustern anordnen

Begriff Rekursion in der Programmierung

Unter **Rekursion** versteht man in der Programmierung eine *Funktion*, die *sich selbst* direkt oder indirekt (über Zwischenaufrufe anderer Funktionen) wieder *aufruft*.

Begriff Rekursion in der Programmierung

Unter **Rekursion** versteht man in der Programmierung eine *Funktion*, die *sich selbst* direkt oder indirekt (über Zwischenaufrufe anderer Funktionen) wieder *aufruft*.

- Üblicherweise *verkleinern* sich mit jedem Selbstaufruf einer Funktion die übergebenen rekursionssteuernden *Parameterwerte*, aber insbesondere dann, wenn die Rekursion über mehrere Parameter läuft, beobachtet man auch zwischenzeitlich steigende Werte.

Begriff Rekursion in der Programmierung

Unter **Rekursion** versteht man in der Programmierung eine *Funktion*, die *sich selbst* direkt oder indirekt (über Zwischenaufrufe anderer Funktionen) wieder *aufruft*.

- Üblicherweise *verkleinern* sich mit jedem Selbstaufruf einer Funktion die übergebenen rekursionssteuernden *Parameterwerte*, aber insbesondere dann, wenn die Rekursion über mehrere Parameter läuft, beobachtet man auch zwischenzeitlich steigende Werte.
- Häufig wird die Berechnung eines Funktionswertes $f(n)$ („großes Problem“) auf die Berechnung des Funktionswertes $f(n-1)$ („kleineres Problem“) zurückgeführt, bis triviale Probleme wie die Berechnung von $f(1)$ oder $f(0)$ entstehen.

Begriff Rekursion in der Programmierung

Unter **Rekursion** versteht man in der Programmierung eine *Funktion*, die *sich selbst* direkt oder indirekt (über Zwischenaufrufe anderer Funktionen) wieder *aufruft*.

- Üblicherweise *verkleinern* sich mit jedem Selbstaufruf einer Funktion die übergebenen rekursionssteuernden *Parameterwerte*, aber insbesondere dann, wenn die Rekursion über mehrere Parameter läuft, beobachtet man auch zwischenzeitlich steigende Werte.
- Häufig wird die Berechnung eines Funktionswertes $f(n)$ („großes Problem“) auf die Berechnung des Funktionswertes $f(n-1)$ („kleineres Problem“) zurückgeführt, bis triviale Probleme wie die Berechnung von $f(1)$ oder $f(0)$ entstehen.

direkter Selbstaufruf (Beispiel): $f(5) \rightarrow f(4) \rightarrow f(3) \rightarrow f(2) \ldots$

indirekter Selbstaufruf (Bsp.): $f(5) \rightarrow g(5) \rightarrow h(5) \rightarrow f(4) \rightarrow g(4) \ldots$

Rekursive Berechnung der Fakultätsfunktion

Aufbereiten der rekursiven Funktionsdefinition für Implementierung

Mathematische Definition führt $f(n+1)$ auf $f(n)$ zurück

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) * n! \end{aligned}$$

Rekursive Berechnung der Fakultätsfunktion

Aufbereiten der rekursiven Funktionsdefinition für Implementierung

Mathematische Definition führt $f(n+1)$ auf $f(n)$ zurück

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) * n! \end{aligned}$$

Programmierung benötigt $f(k) = \ldots$, ersetze $n+1$ durch k

$$\begin{aligned} 0! &= 1 \\ k! &= k * (k-1)! \end{aligned}$$

Rekursive Berechnung der Fakultätsfunktion

Aufbereiten der rekursiven Funktionsdefinition für Implementierung

Mathematische Definition führt $f(n+1)$ auf $f(n)$ zurück

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) * n! \end{aligned}$$

Programmierung benötigt $f(k) = \ldots$, ersetze $n+1$ durch k

$$\begin{aligned} 0! &= 1 \\ k! &= k * (k-1)! \end{aligned}$$

Funktionsnamen in *Präfixschreibweise* notieren

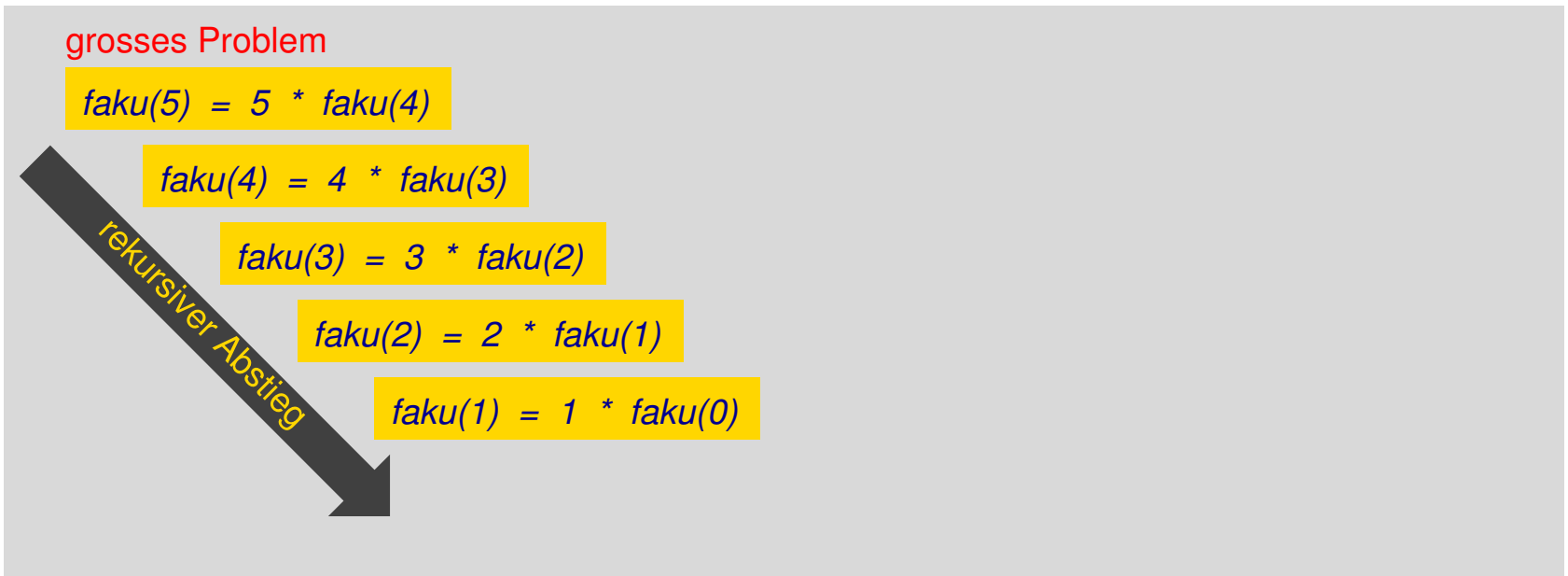
$$\begin{aligned} faku(0) &= 1 \\ faku(k) &= k * faku(k-1) \end{aligned}$$

Rekursive Berechnung der Fakultätsfunktion

Drei Phasen der Rekursionsabarbeitung

$$\begin{aligned} faku(0) &= 1 \\ faku(k) &= k * faku(k-1) \end{aligned}$$

Im *rekursiven Abstieg* Problem schrittweise verkleinert ...

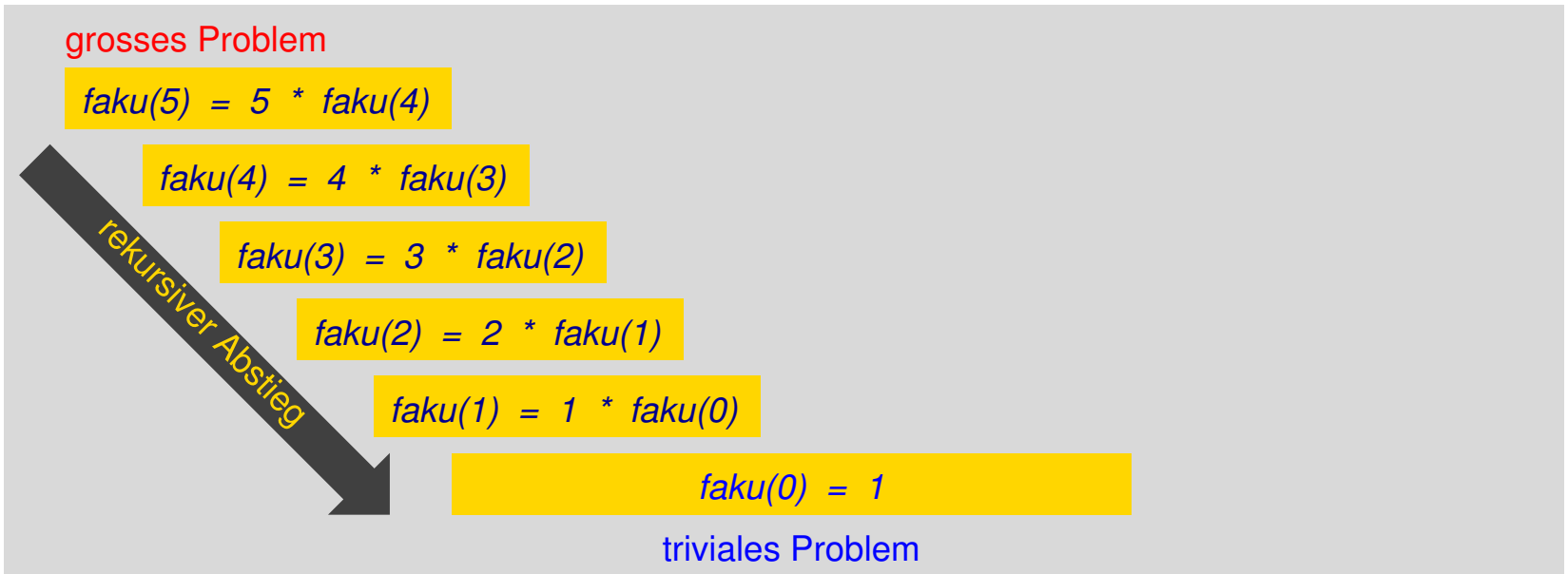


Rekursive Berechnung der Fakultätsfunktion

Drei Phasen der Rekursionsabarbeitung

$$\begin{aligned} faku(0) &= 1 \\ faku(k) &= k * faku(k-1) \end{aligned}$$

. . . bis Problem *trivial* und Lösung direkt angebbar

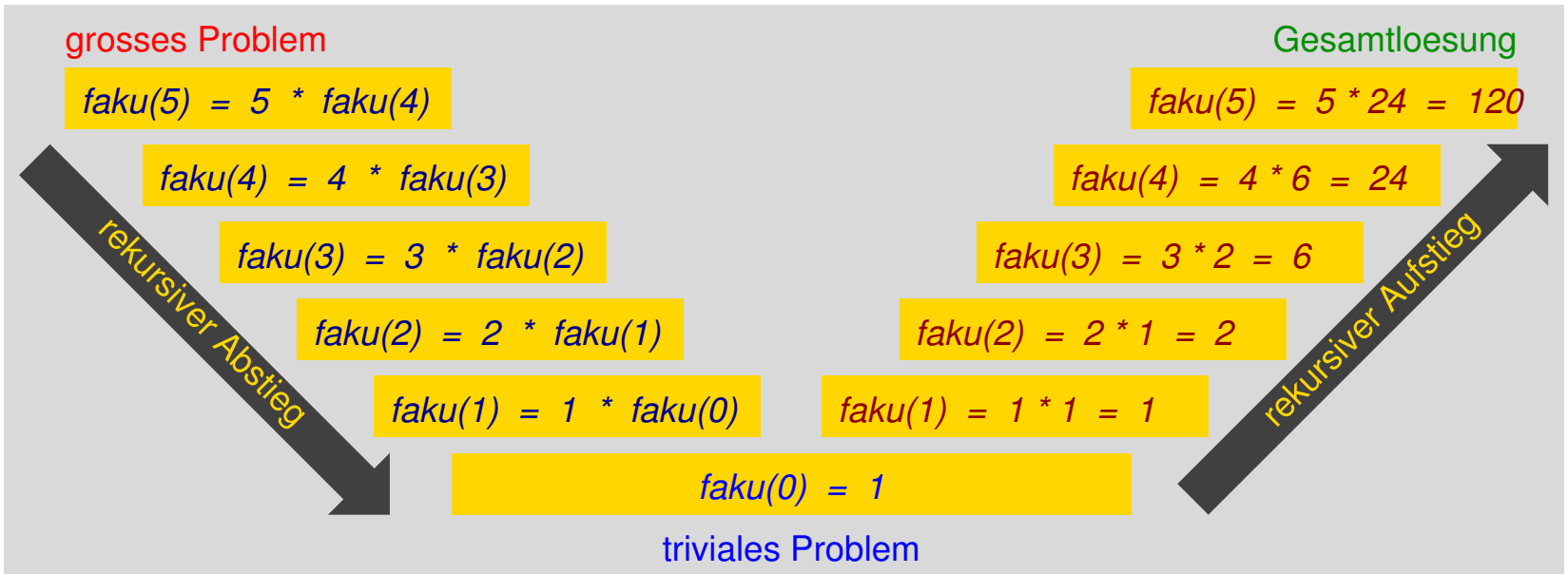


Rekursive Berechnung der Fakultätsfunktion

Drei Phasen der Rekursionsabarbeitung

$$\begin{aligned} faku(0) &= 1 \\ faku(k) &= k * faku(k-1) \end{aligned}$$

Im *rekursiven Aufstieg* schrittweiser Aufbau der Gesamtlösung

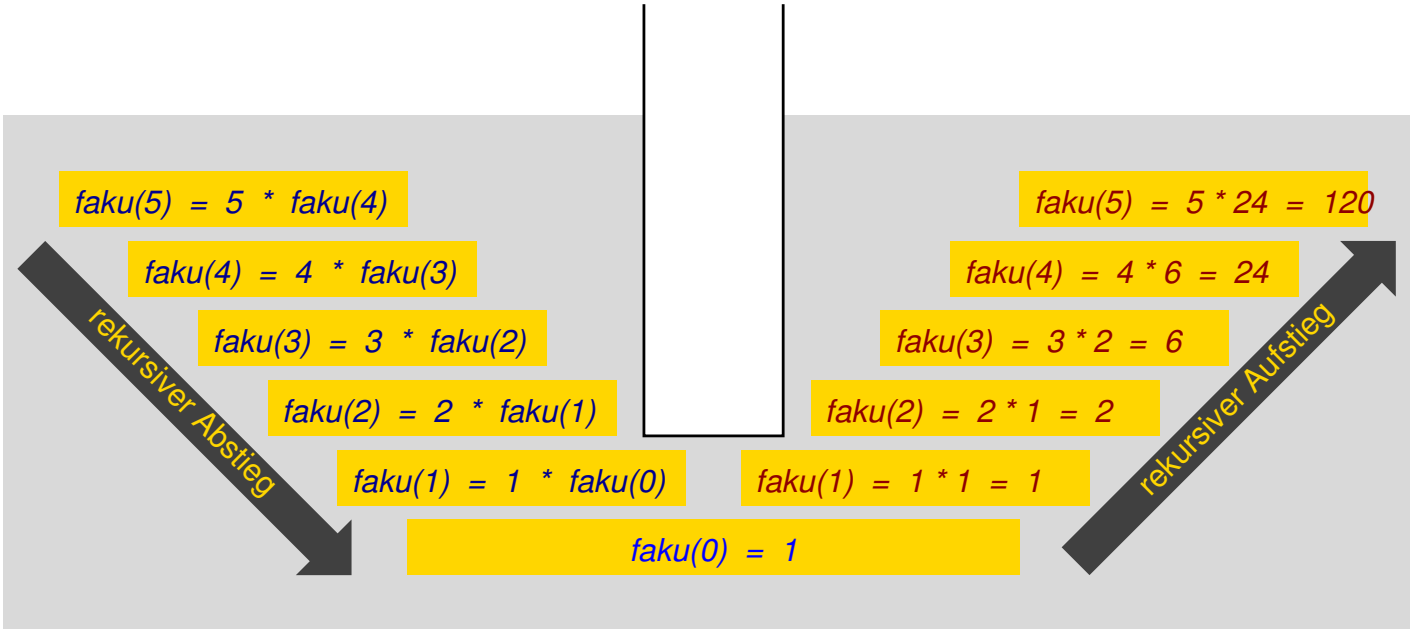


Rekursive Berechnung der Fakultätsfunktion (faku.c)

```c
#include <stdio.h>

unsigned long faku(int k)
{
  if (k == 0)
  {
    return 1;
  }
  return k * faku(k-1);
}


int main(void)
{
  int n;

  printf("\nBerechnung von n!. Bitte n eingeben: ");
  scanf("%d", &n);
  if (n >= 0)
  {
    printf("%d! ist %lu\n", n, faku(n));
  }
  return 0;
}
```

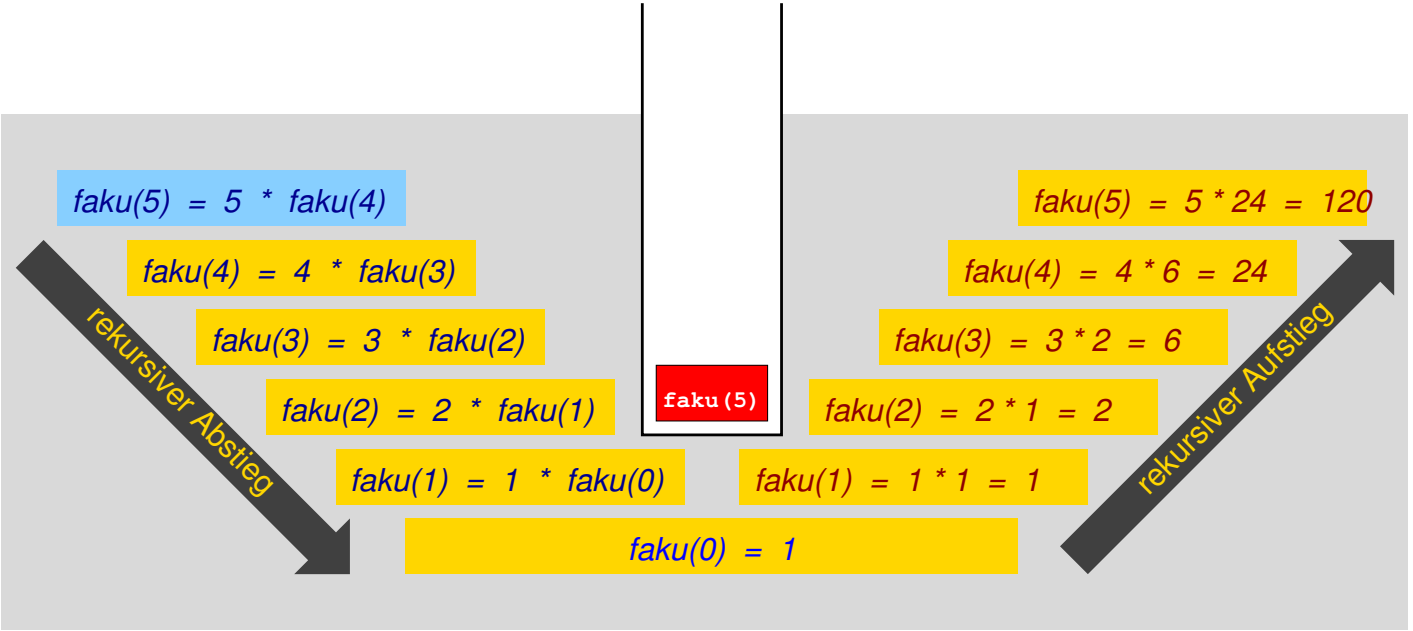
$12! = 479\,001\,600$ liegt noch im Wertebereich von **`unsigned long`**

Funktionsaufrufstack zur Rekursionsabarbeitung



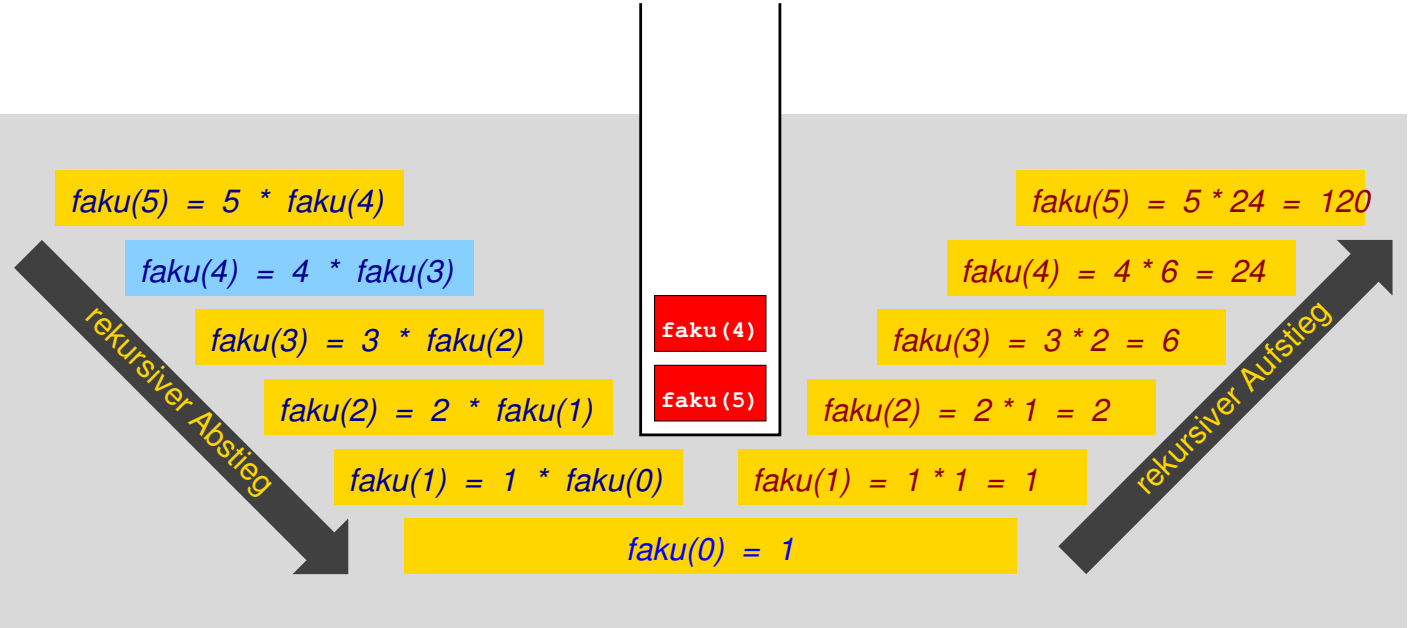
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



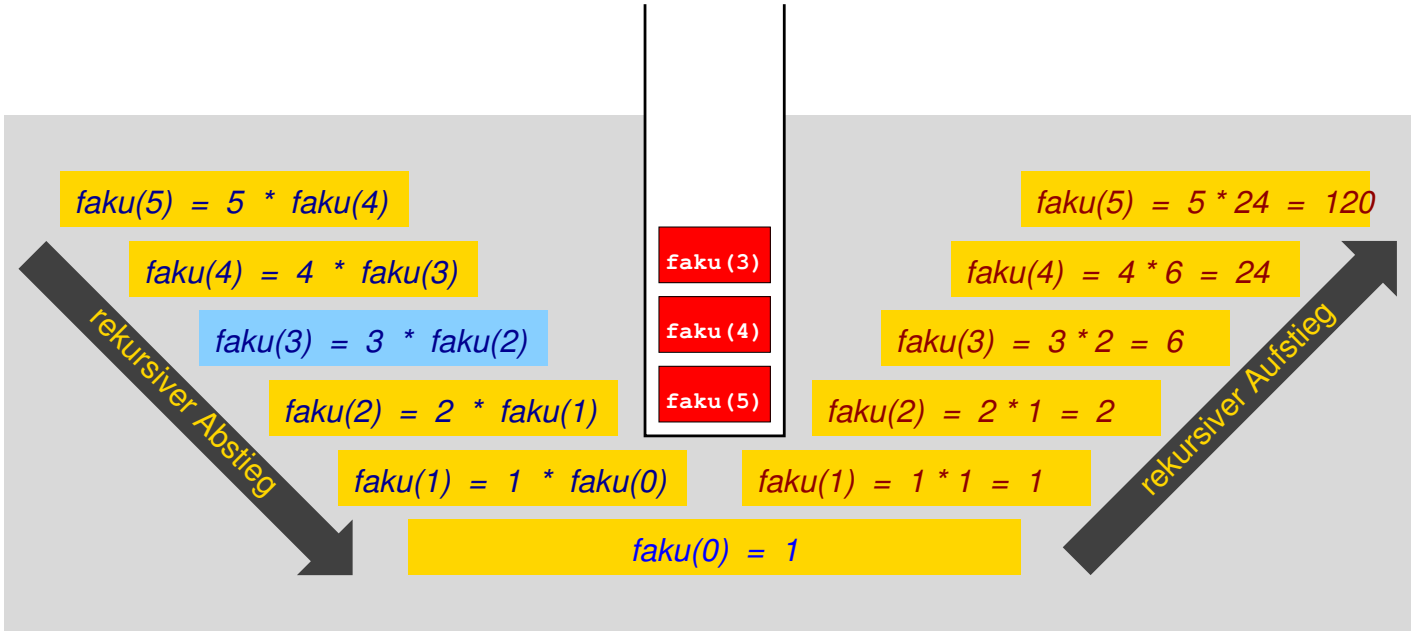
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



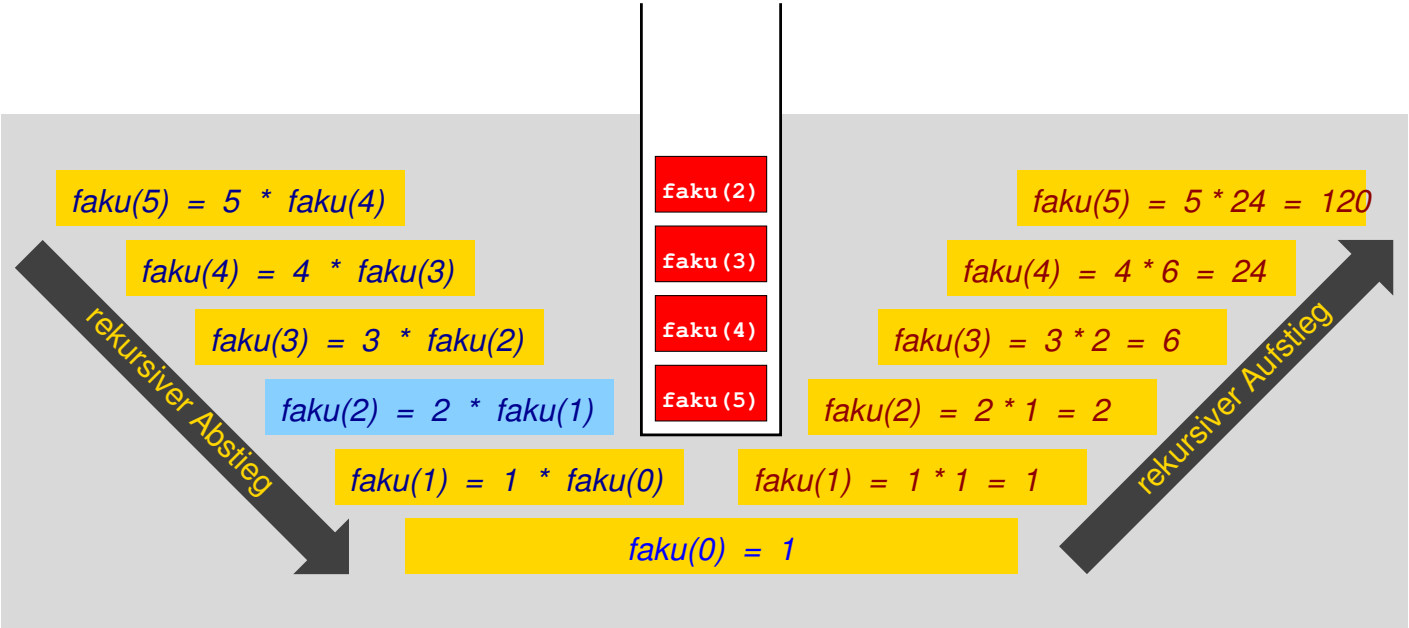
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



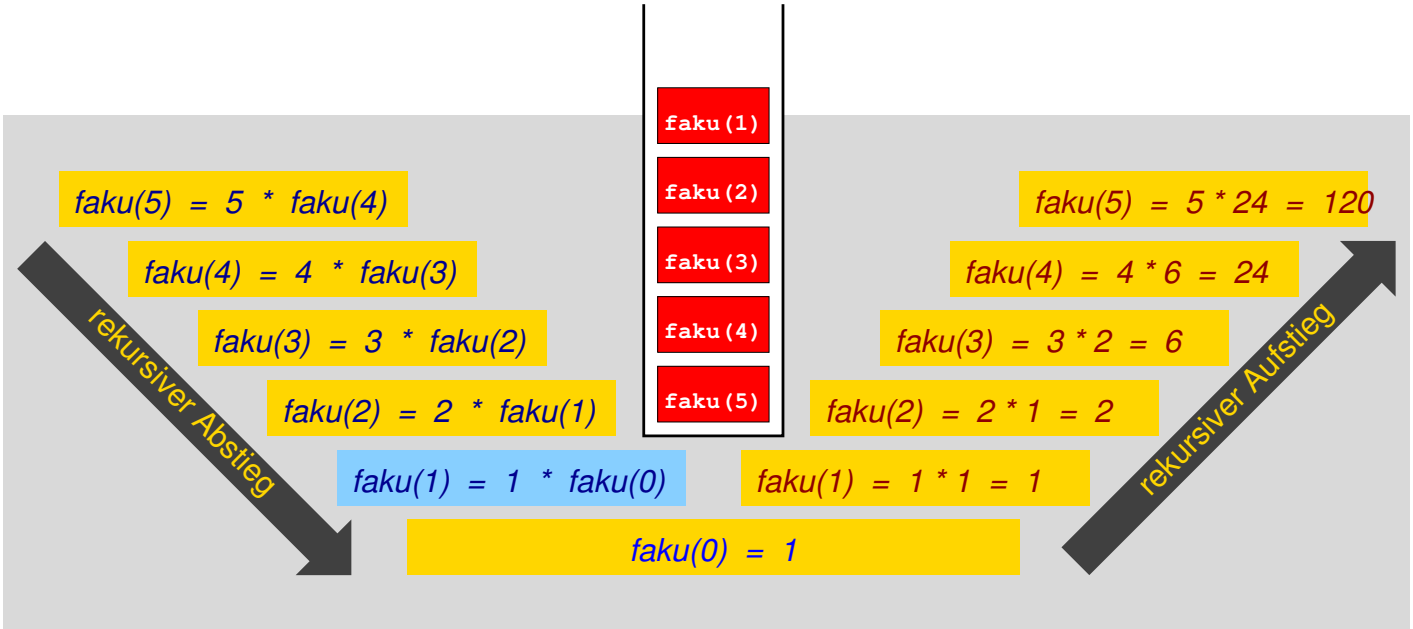
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



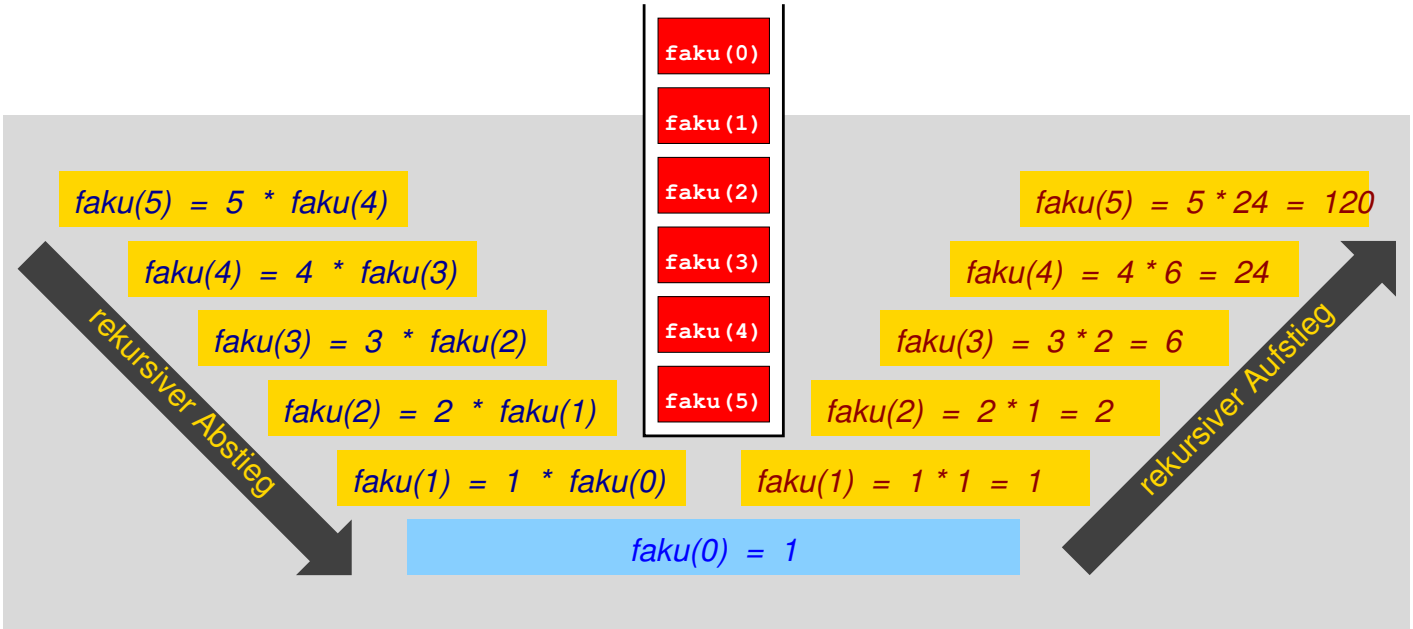
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



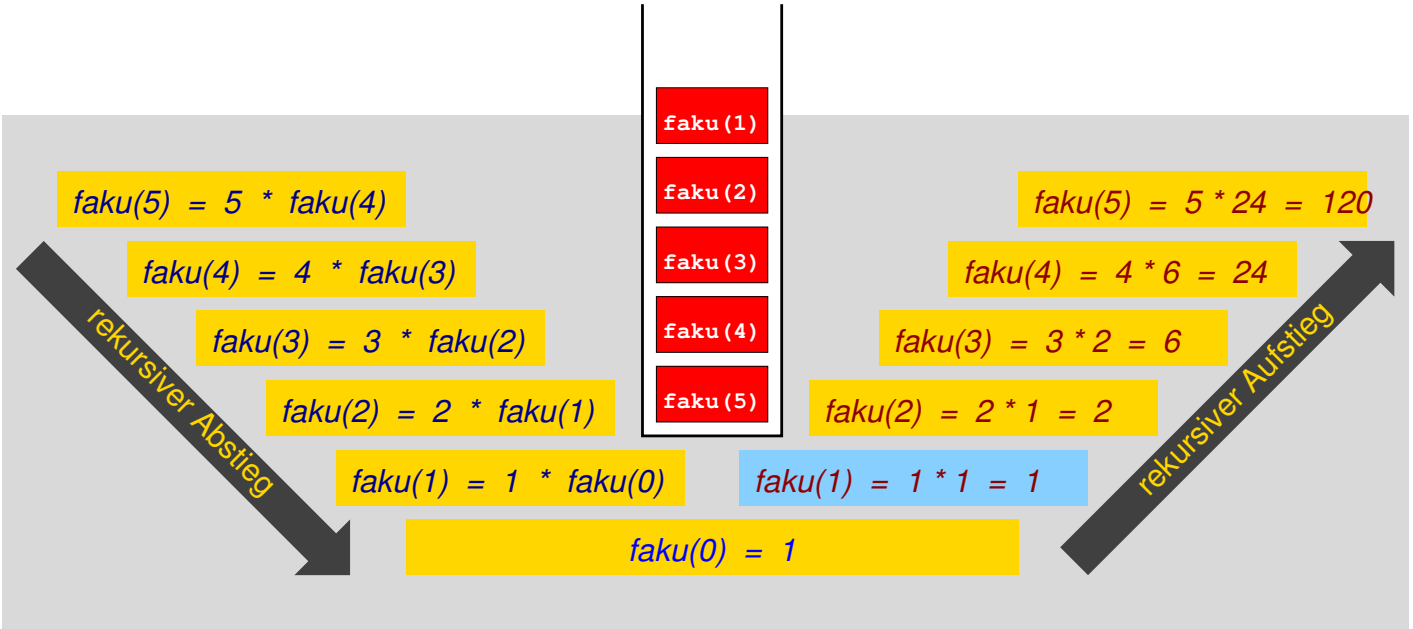
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



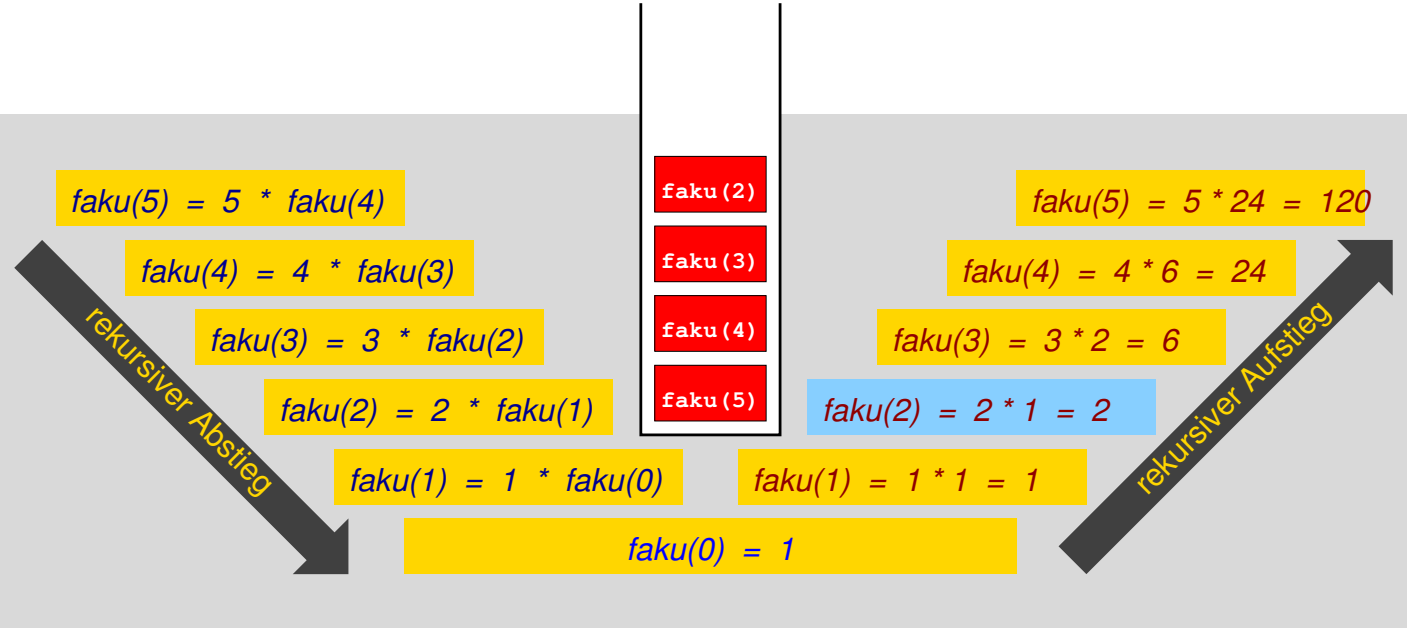
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



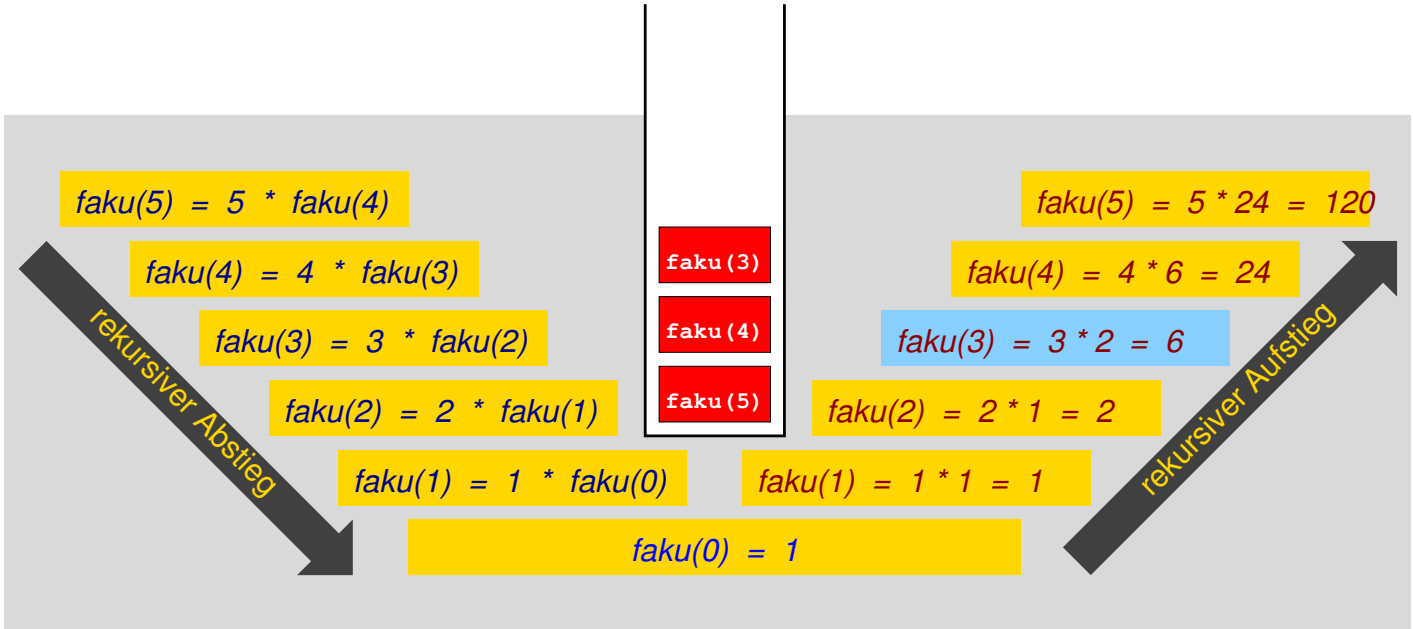
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



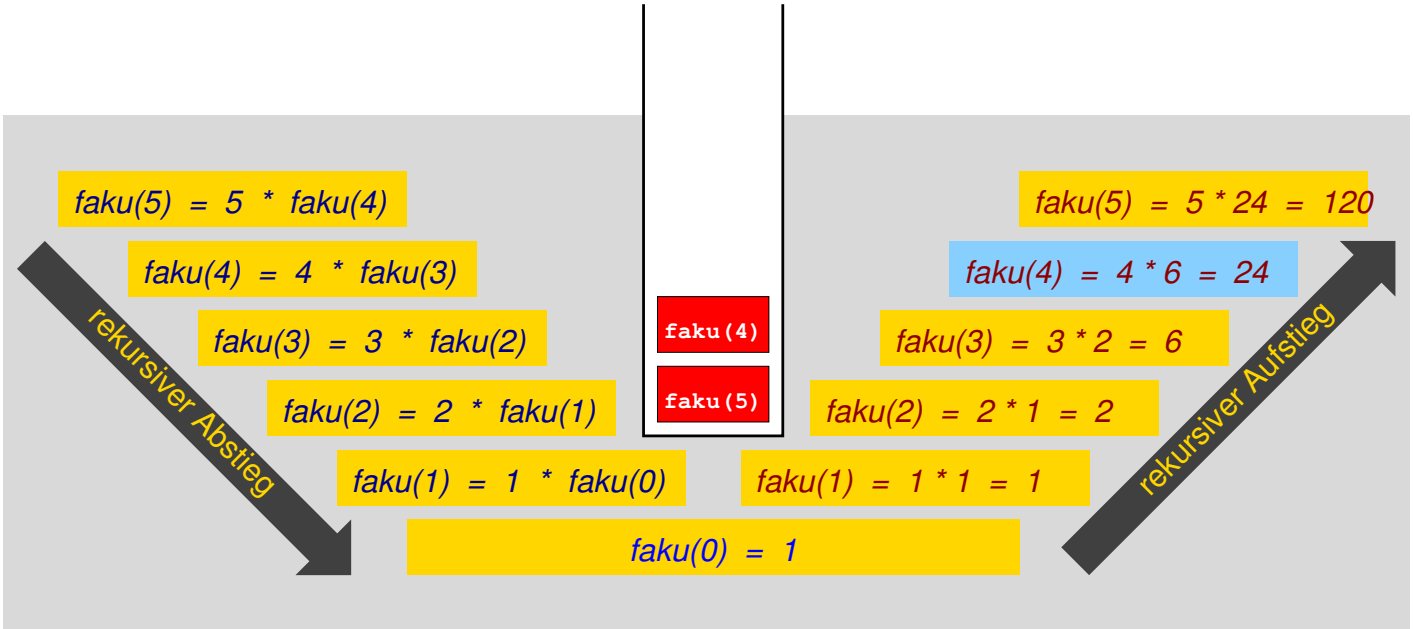
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



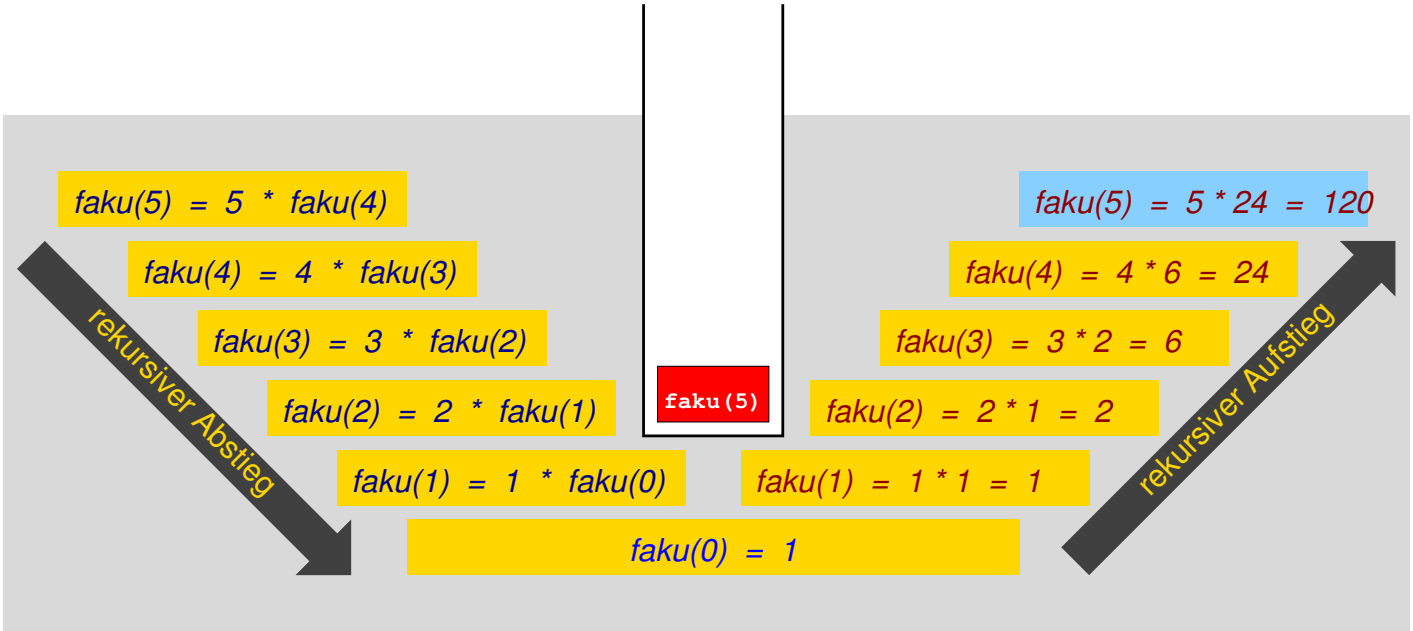
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



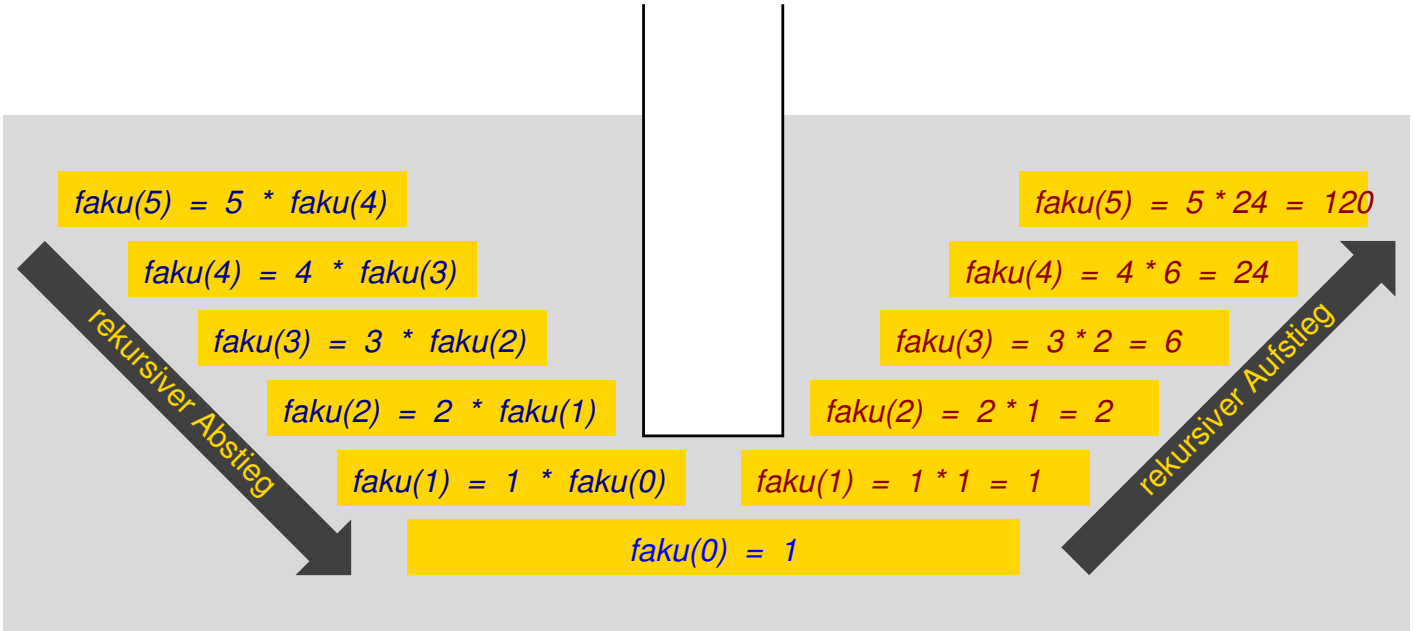
Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Funktionsaufrufstack zur Rekursionsabarbeitung



Der ***Funktionsaufrufstack*** ist ein in seiner Größe variabler Speicherbereich, der nach dem Prinzip *„Last In First Out (LIFO)“* organisiert ist. Mit *jedem* neuen *Funktionsaufruf* während des rekursiven Abstiegs werden die aktuellen *Argumentwerte* und die *Rücksprungadresse* zur Fortsetzung des Maschinenprogramms nach Aufrufabarbeitung *gespeichert*. Im rekursiven Aufstieg leert sich der Funktionsaufrufstack schrittweise.

Eine Bibliothek rekursiver arithmetischer Funktionen

pre, add, mul, pow2, nsub, div, ld auf natürlichen Zahlen – rekarith.c

Vorgängerfunktion *pre* : $\mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} pre(0) &= 0 \\ pre(n+1) &= n \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

pre, add, mul, pow2, nsub, div, ld auf natürlichen Zahlen – rekarith.c

Vorgängerfunktion *pre* : $\mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} pre(0) &= 0 \\ pre(n+1) &= n \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} pre(0) &= 0 \\ pre(k) &= k \dot{-} 1 \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

pre, add, mul, pow2, nsub, div, ld auf natürlichen Zahlen – rekarith.c

Vorgängerfunktion *pre* : $\mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} pre(0) &= 0 \\ pre(n+1) &= n \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} pre(0) &= 0 \\ pre(k) &= k \dot{-} 1 \end{aligned}$$

```c
/* Vorgaengerfunktion */

unsigned long pre(unsigned long k)
{
  if (k == 0)
  {
    return 0;
  }
  return k-1;
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

pre, add, mul, pow2, nsub, div, ld auf natürlichen Zahlen – rekarith.c

Vorgängerfunktion *pre* : $\mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} pre(0) &= 0 \\ pre(n+1) &= n \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} pre(0) &= 0 \\ pre(k) &= k \dot{-} 1 \end{aligned}$$

Beispiel

$pre(2) = 1$

```c
/* Vorgaengerfunktion */

unsigned long pre(unsigned long k)
{
  if (k == 0)
  {
    return 0;
  }
  return k-1;
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Addition *add* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 0 + x &= x \\ (n+1) + x &= (n+x) + 1 \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Addition *add* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 0 + x &= x \\ (n+1) + x &= (n+x) + 1 \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} add(0, x) &= x \\ add(k, x) &= add(pre(k), x) + 1 \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Addition *add* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 0 + x &= x \\ (n+1) + x &= (n+x) + 1 \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} add(0, x) &= x \\ add(k, x) &= add(pre(k), x) + 1 \end{aligned}$$

```
/* Addition */

unsigned long add(unsigned long k,
                  unsigned long x)
{
  if (k == 0)
  {
    return x;
  }
  return add(pre(k), x) + 1;
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Addition *add* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 0 + x &= x \\ (n+1) + x &= (n+x) + 1 \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} add(0, x) &= x \\ add(k, x) &= add(pre(k), x) + 1 \end{aligned}$$

Beispiel

$$\begin{aligned} add(2,3) &= add(1,3) + 1 \\ &= add(0,3) + 1 + 1 \\ &= 3 + 1 + 1 \\ &= 5 \end{aligned}$$

```
/* Addition */

unsigned long add(unsigned long k,
                  unsigned long x)
{
  if (k == 0)
  {
    return x;
  }
  return add(pre(k), x) + 1;
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Multiplikation *mul* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 0 \cdot x &= 0 \\ (n+1) \cdot x &= n \cdot x + x \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Multiplikation *mul* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 0 \cdot x &= 0 \\ (n+1) \cdot x &= n \cdot x + x \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} mul(0, x) &= 0 \\ mul(k, x) &= add(x, mul(pre(k), x)) \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Multiplikation *mul* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 0 \cdot x &= 0 \\ (n+1) \cdot x &= n \cdot x + x \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} mul(0, x) &= 0 \\ mul(k, x) &= add(x, mul(pre(k), x)) \end{aligned}$$

```
/* Multiplikation */

unsigned long mul(unsigned long k,
                  unsigned long x)
{
  if (k == 0)
  {
    return 0;
  }
  return add(x,mul(pre(k), x));
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Multiplikation *mul* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 0 \cdot x &= 0 \\ (n+1) \cdot x &= n \cdot x + x \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} mul(0, x) &= 0 \\ mul(k, x) &= add(x, mul(pre(k), x)) \end{aligned}$$

Beispiel

$$\begin{aligned} mul(2,3) &= add(3, mul(1,3)) \\ &= add(3, add(3, mul(0,3))) \\ &= add(3, add(3,0)) \\ &= add(3,3) = 6 \end{aligned}$$

```
/* Multiplikation */

unsigned long mul(unsigned long k,
                  unsigned long x)
{
  if (k == 0)
  {
    return 0;
  }
  return add(x,mul(pre(k), x));
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Zweierpotenz *pow2* : $\mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 2^0 &= 1 \\ 2^{n+1} &= 2 \cdot 2^n \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Zweierpotenz *pow2* : $\mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 2^0 &= 1 \\ 2^{n+1} &= 2 \cdot 2^n \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} pow2(0) &= 1 \\ pow2(k) &= mul(2, pow2(pre(k))) \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Zweierpotenz *pow2* : $\mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 2^0 &= 1 \\ 2^{n+1} &= 2 \cdot 2^n \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} pow2(0) &= 1 \\ pow2(k) &= mul(2, pow2(pre(k))) \end{aligned}$$

```
/* Zweierpotenz */

unsigned long pow2(unsigned long k)
{
  if (k == 0)
  {
    return 1;
  }
  return mul(2,pow2(pre(k)));
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Zweierpotenz *pow2* : $\mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 2^0 &= 1 \\ 2^{n+1} &= 2 \cdot 2^n \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} pow2(0) &= 1 \\ pow2(k) &= mul(2, pow2(pre(k))) \end{aligned}$$

Beispiel

$$\begin{aligned} pow2(3) &= mul(2, pow2(2)) \\ &= mul(2, mul(2, pow2(1))) \\ &= mul(2, mul(2, mul(2, pow2(0)))) \\ &= mul(2, mul(2, mul(2, 1))) = 8 \end{aligned}$$

```
/* Zweierpotenz */

unsigned long pow2(unsigned long k)
{
  if (k == 0)
  {
    return 1;
  }
  return mul(2,pow2(pre(k)));
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Zweierpotenz *pow2* : $\mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} 2^0 &= 1 \\ 2^{n+1} &= 2 \cdot 2^n \end{aligned}$$

Unvorteilhafte Implementierung

$$\begin{aligned} 2^0 &= 1 \\ 2^{n+1} &= 2^n + 2^n \end{aligned}$$

Jeder nichttriviale Funktionsaufruf zieht hier *zwei* Folgeaufrufe nach sich, wodurch *exponentielle* Anzahl von Aufrufen entsteht. Zudem werden gleiche Funktionswerte mehrfach neu berechnet (Redundanz).

Eine Bibliothek rekursiver arithmetischer Funktionen

Nichtnegative Subtraktion *nsub* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$
\begin{aligned}
x \dot{-} 0 &= x \\
x \dot{-} (n+1) &= (x \dot{-} n) \dot{-} 1
\end{aligned}
$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Nichtnegative Subtraktion *nsub* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$
\begin{aligned}
x \dot{-} 0 &= x \\
x \dot{-} (n+1) &= (x \dot{-} n) \dot{-} 1
\end{aligned}
$$

Ersetzung $k = n + 1$

$$
\begin{aligned}
nsub(0, x) &= x \\
nsub(k, x) &= pre(nsub(pre(k), x))
\end{aligned}
$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Nichtnegative Subtraktion *nsub* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$
\begin{aligned}
x \dot{-} 0 &= x \\
x \dot{-} (n+1) &= (x \dot{-} n) \dot{-} 1
\end{aligned}
$$

Ersetzung $k = n + 1$

$$
\begin{aligned}
nsub(0, x) &= x \\
nsub(k, x) &= pre(nsub(pre(k), x))
\end{aligned}
$$

```c
/* Nichtnegative Subtraktion */

unsigned long nsub(unsigned long k,
                   unsigned long x)
{
  if (k == 0)
  {
    return x;
  }
  return pre(nsub(pre(k),x));
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Nichtnegative Subtraktion *nsub* : $\mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (n+1) &= (x \dot{-} n) \dot{-} 1 \end{aligned}$$

Ersetzung $k = n + 1$

$$\begin{aligned} nsub(0, x) &= x \\ nsub(k, x) &= pre(nsub(pre(k), x)) \end{aligned}$$

```
/* Nichtnegative Subtraktion */

unsigned long nsub(unsigned long k,
                   unsigned long x)
{
  if (k == 0)
  {
    return x;
  }
  return pre(nsub(pre(k),x));
}
```

Beispiel

$$\begin{aligned} nsub(3,7) &= pre(nsub(2,7)) \\ &= pre(pre(nsub(1,7))) \\ &= pre(pre(pre(nsub(0,7)))) \\ &= pre(pre(pre(7))) = 4 \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Division *div* : $\mathbb{N} \times \mathbb{N}_+ \longrightarrow \mathbb{N}$

$$\left\lfloor \frac{x}{n} \right\rfloor = 0 \text{ falls } x < n$$

$$\left\lfloor \frac{x+n}{n} \right\rfloor = \left\lfloor \frac{x}{n} \right\rfloor + 1 \text{ sonst}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Division *div* : $\mathbb{N} \times \mathbb{N}_+ \longrightarrow \mathbb{N}$

$$\left\lfloor \frac{x}{n} \right\rfloor = 0 \text{ falls } x < n$$

$$\left\lfloor \frac{x+n}{n} \right\rfloor = \left\lfloor \frac{x}{n} \right\rfloor + 1 \text{ sonst}$$

Ersetzung zu $\left\lfloor \frac{x}{k} \right\rfloor = \left\lfloor \frac{x-k}{k} \right\rfloor + 1$

$$div(k, x) = 0 \text{ falls } x < k$$
$$div(k, x) = add(1, div(k, nsub(k, x)))$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Division *div* : $\mathbb{N} \times \mathbb{N}_+ \longrightarrow \mathbb{N}$

$$\left\lfloor \frac{x}{n} \right\rfloor = 0 \text{ falls } x < n$$

$$\left\lfloor \frac{x+n}{n} \right\rfloor = \left\lfloor \frac{x}{n} \right\rfloor + 1 \text{ sonst}$$

Ersetzung zu $\left\lfloor \frac{x}{k} \right\rfloor = \left\lfloor \frac{x-k}{k} \right\rfloor + 1$

$$div(k, x) = 0 \text{ falls } x < k$$

$$div(k, x) = add(1, div(k, nsub(k, x)))$$

```
/* Division */

unsigned long div(unsigned long k,
                  unsigned long x)
{
  if (x < k)
  {
    return 0;
  }
  return add(1,div(k,nsub(k,x)));
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Division *div* : $\mathbb{N} \times \mathbb{N}_+ \longrightarrow \mathbb{N}$

$$\left\lfloor \frac{x}{n} \right\rfloor = 0 \text{ falls } x < n$$

$$\left\lfloor \frac{x+n}{n} \right\rfloor = \left\lfloor \frac{x}{n} \right\rfloor + 1 \text{ sonst}$$

```
/* Division */

unsigned long div(unsigned long k,
                  unsigned long x)
{
  if (x < k)
  {
    return 0;
  }
  return add(1,div(k,nsub(k,x)));
}
```

Ersetzung zu $\left\lfloor \frac{x}{k} \right\rfloor = \left\lfloor \frac{x-k}{k} \right\rfloor + 1$

$$div(k,x) = 0 \text{ falls } x < k$$

$$div(k,x) = add(1, div(k, nsub(k,x)))$$

Beispiel

$$\begin{aligned} div(3,8) &= add(1, div(3,5)) \\ &= add(1, add(1, div(3,2))) \\ &= add(1, add(1,0)) \\ &= 2 \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Logarithmus dualis *ld* : $\mathbb{N}_+ \longrightarrow \mathbb{N}$

$$\begin{aligned} \mathrm{ld}(1) &= 0 \\ \mathrm{ld}(2 \cdot n) &= \mathrm{ld}(n) + 1 \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Logarithmus dualis *ld* : $\mathbb{N}_+ \longrightarrow \mathbb{N}$

$$\begin{aligned} \mathrm{ld}(1) &= 0 \\ \mathrm{ld}(2 \cdot n) &= \mathrm{ld}(n) + 1 \end{aligned}$$

Ersetzung $k = 2n$

$$\begin{aligned} ld(1) &= 0 \\ ld(k) &= add(1, ld(div(2, k))) \end{aligned}$$

Eine Bibliothek rekursiver arithmetischer Funktionen

Logarithmus dualis *ld* : $\mathbb{N}_+ \longrightarrow \mathbb{N}$

$$\begin{aligned} \mathrm{ld}(1) &= 0 \\ \mathrm{ld}(2 \cdot n) &= \mathrm{ld}(n) + 1 \end{aligned}$$

Ersetzung $k = 2n$

$$\begin{aligned} ld(1) &= 0 \\ ld(k) &= add(1, ld(div(2, k))) \end{aligned}$$

```
/* Logarithmus dualis */

unsigned long ld(unsigned long k)
{
  if (k == 1)
  {
    return 0;
  }
  return add(1,ld(div(2,k)));
}
```

Eine Bibliothek rekursiver arithmetischer Funktionen

Logarithmus dualis *ld* : $\mathbb{N}_+ \longrightarrow \mathbb{N}$

$$\begin{aligned} \mathrm{ld}(1) &= 0 \\ \mathrm{ld}(2 \cdot n) &= \mathrm{ld}(n) + 1 \end{aligned}$$

```
/* Logarithmus dualis */

unsigned long ld(unsigned long k)
{
  if (k == 1)
  {
    return 0;
  }
  return add(1,ld(div(2,k)));
}
```

Ersetzung $k = 2n$

$$\begin{aligned} ld(1) &= 0 \\ ld(k) &= add(1, ld(div(2, k))) \end{aligned}$$

Beispiel

$$\begin{aligned} ld(18) &= add(1, ld(9)) \\ &= add(1, add(1, ld(4))) \\ &= add(1, add(1, add(1, ld(2)))) \\ &= add(1, add(1, add(1, add(1, ld(1))))) \\ &= add(1, add(1, add(1, add(1, 0)))) = 4 \end{aligned}$$

Berechnung eines umfangreichen Terms (rekarith.c)

$$\mathrm{ld}\left(5 \cdot (2+4) \dot{-} \left\lfloor \frac{7}{3} \right\rfloor\right)$$

```c
int main(void)
{
  printf("Berechnung ld(5*(2+4) - 7/3) = ");
  printf("%lu\n", ld(nsub(div(3,7),mul(5,add(2,4)))));
  return 0;
}
```

$ld(nsub(div(3,7), mul(5, add(2,4))))$

Funktionsaufrufstack *pulsiert* während Berechnungsvorgang

Berechnung eines umfangreichen Terms (rekarith.c)

$$\mathrm{ld}\left(5 \cdot (2+4) \dot{-} \left\lfloor \frac{7}{3} \right\rfloor\right)$$

```c
int main(void)
{
  printf("Berechnung ld(5*(2+4) - 7/3) = ");
  printf("%lu\n", ld(nsub(div(3,7),mul(5,add(2,4)))));
  return 0;
}
```

$$ld(nsub(div(3,7), mul(5, add(2,4)))) = ld(nsub(2, mul(5, add(2,4))))$$

Funktionsaufrufstack *pulsiert* während Berechnungsvorgang

Berechnung eines umfangreichen Terms (rekarith.c)

$$\mathrm{ld}\left(5 \cdot (2+4) \dot{-} \left\lfloor \frac{7}{3} \right\rfloor\right)$$

```c
int main(void)
{
  printf("Berechnung ld(5*(2+4) - 7/3) = ");
  printf("%lu\n", ld(nsub(div(3,7),mul(5,add(2,4)))));
  return 0;
}
```

$$\begin{aligned} &ld(nsub(div(3,7), mul(5, add(2,4)))) \\ &\quad = ld(nsub(2, mul(5, add(2,4)))) \\ &\quad = ld(nsub(2, mul(5,6))) \end{aligned}$$

Funktionsaufrufstack *pulsiert* während Berechnungsvorgang

Berechnung eines umfangreichen Terms (rekarith.c)

$$\mathrm{ld}\left(5 \cdot (2+4) \dot{-} \left\lfloor \frac{7}{3} \right\rfloor\right)$$

```c
int main(void)
{
  printf("Berechnung ld(5*(2+4) - 7/3) = ");
  printf("%lu\n", ld(nsub(div(3,7),mul(5,add(2,4)))));
  return 0;
}
```

$$\begin{aligned} &ld(nsub(div(3,7), mul(5, add(2,4)))) \\ &\quad = ld(nsub(2, mul(5, add(2,4)))) \\ &\quad = ld(nsub(2, mul(5,6))) \\ &\quad = ld(nsub(2,30)) \end{aligned}$$

Funktionsaufrufstack *pulsiert* während Berechnungsvorgang

Berechnung eines umfangreichen Terms (rekarith.c)

$$\mathrm{ld}\left(5 \cdot (2+4) \dot{-} \left\lfloor \frac{7}{3} \right\rfloor\right)$$

```c
int main(void)
{
  printf("Berechnung ld(5*(2+4) - 7/3) = ");
  printf("%lu\n", ld(nsub(div(3,7),mul(5,add(2,4)))));
  return 0;
}
```

$$\begin{aligned} & ld(nsub(div(3,7), mul(5, add(2,4)))) \\ &= ld(nsub(2, mul(5, add(2,4)))) \\ &= ld(nsub(2, mul(5,6))) \\ &= ld(nsub(2,30)) \\ &= ld(28) \end{aligned}$$

Funktionsaufrufstack *pulsiert* während Berechnungsvorgang

Berechnung eines umfangreichen Terms (rekarith.c)

$$\mathrm{ld}\left(5 \cdot (2+4) \dot{-} \left\lfloor \frac{7}{3} \right\rfloor\right)$$

```c
int main(void)
{
  printf("Berechnung ld(5*(2+4) - 7/3) = ");
  printf("%lu\n", ld(nsub(div(3,7),mul(5,add(2,4)))));
  return 0;
}
```

$$\begin{aligned} & ld(nsub(div(3,7), mul(5, add(2,4)))) \\ &= ld(nsub(2, mul(5, add(2,4)))) \\ &= ld(nsub(2, mul(5,6))) \\ &= ld(nsub(2,30)) \\ &= ld(28) \\ &= 4 \end{aligned}$$

Funktionsaufrufstack *pulsiert* während Berechnungsvorgang

Wechselgeld optimal stückeln – rekursiv programmiert

Geldbetrag in möglichst wenigen Scheinen/Münzen darstellen

Beispiel: 546.28 entspricht

$$500.00 + 20.00 + 20.00 + 5.00 + 1.00 + 0.20 + 0.05 + 0.02 + 0.01$$

Rekursiver Greedy-Algorithmus (wechselgeld.c)

Geldbetrag in möglichst wenigen Scheinen/Münzen darstellen

```c
#include <stdio.h>

unsigned long geldeinheiten(unsigned long k) /* k ist Centwert */
{
  if (k >= 50000) {printf("500.00  "); return geldeinheiten(k-50000);}
  if (k >= 20000) {printf("200.00  "); return geldeinheiten(k-20000);}
  if (k >= 10000) {printf("100.00  "); return geldeinheiten(k-10000);}
  if (k >= 5000) {printf("50.00  "); return geldeinheiten(k-5000);}
  if (k >= 2000) {printf("20.00  "); return geldeinheiten(k-2000);}
  if (k >= 1000) {printf("10.00  "); return geldeinheiten(k-1000);}
  if (k >= 500) {printf("5.00  "); return geldeinheiten(k-500);}
  if (k >= 200) {printf("2.00  "); return geldeinheiten(k-200);}
  if (k >= 100) {printf("1.00  "); return geldeinheiten(k-100);}
  if (k >= 50) {printf("0.50  "); return geldeinheiten(k-50);}
  if (k >= 20) {printf("0.20  "); return geldeinheiten(k-20);}
  if (k >= 10) {printf("0.10  "); return geldeinheiten(k-10);}
  if (k >= 5) {printf("0.05  "); return geldeinheiten(k-5);}
  if (k >= 2) {printf("0.02  "); return geldeinheiten(k-2);}
  if (k >= 1) {printf("0.01  "); return geldeinheiten(k-1);}
  printf("\n");
  return 0;
}


int main(void)
{
  float geldbetrag;

  printf("Bitte Geldbetrag eingeben: ");
  scanf("%f", &geldbetrag);
  if (geldbetrag > 0) {geldeinheiten((unsigned long) (geldbetrag*100));}
  return 0;
}
```

```
Bitte Geldbetrag eingeben: 546.28
500.00  20.00  20.00  5.00  1.00  0.20  0.05  0.02  0.01
```

Rekursion versus Iteration

Jede *Rekursion* lässt sich semantisch äquivalent durch *Schleifen (iterativ)* nachbilden und umgekehrt.

Rekursion versus Iteration

Jede *Rekursion* lässt sich semantisch äquivalent durch *Schleifen (iterativ)* nachbilden und umgekehrt.
Beide Konzepte sind *gleichwertig*, was ihre Beschreibungsmächtigkeit angeht.

Rekursion versus Iteration

Jede *Rekursion* lässt sich semantisch äquivalent durch *Schleifen (iterativ)* nachbilden und umgekehrt.
Beide Konzepte sind *gleichwertig*, was ihre Beschreibungsmächtigkeit angeht.

- Schleifenimplementierungen haben Vorteile bei der technischen Umsetzung (weniger Hilfsspeicherplatz benötigt, kein Funktionsaufrufstack),
- während Rekursion näher an der Mathematik ist und beweisbar korrekte Implementierungen erlaubt.
- Es bietet sich an, *Rekursionen in Iterationen umzuwandeln*.

Wechselgeld-Programm iterativ (wechselgeld-iterativ.c)

`while`-Schleife statt Kette rekursiver Aufrufe zum Runterzählen von `k`

```c
#include <stdio.h>

unsigned long geldeinheiten(unsigned long k) /* k ist Centwert */
{
  while (k > 0)
  {
    if (k >= 50000) {printf("500.00  "); k = k-50000; continue;}
    if (k >= 20000) {printf("200.00  "); k = k-20000; continue;}
    if (k >= 10000) {printf("100.00  "); k = k-10000; continue;}
    if (k >= 5000) {printf("50.00  "); k = k-5000; continue;}
    if (k >= 2000) {printf("20.00  "); k = k-2000; continue;}
    if (k >= 1000) {printf("10.00  "); k = k-1000; continue;}
    if (k >= 500) {printf("5.00  "); k = k-500; continue;}
    if (k >= 200) {printf("2.00  "); k = k-200; continue;}
    if (k >= 100) {printf("1.00  "); k = k-100; continue;}
    if (k >= 50) {printf("0.50  "); k = k-50; continue;}
    if (k >= 20) {printf("0.20  "); k = k-20; continue;}
    if (k >= 10) {printf("0.10  "); k = k-10; continue;}
    if (k >= 5) {printf("0.05  "); k = k-5; continue;}
    if (k >= 2) {printf("0.02  "); k = k-2; continue;}
    if (k >= 1) {printf("0.01  "); k = k-1; continue;}
  }
  printf("\n");
  return 0;
}
```

```
Bitte Geldbetrag eingeben: 546.28
500.00  20.00  20.00  5.00  1.00  0.20  0.05  0.02  0.01
```

```c
int main(void)
{
  float geldbetrag;

  printf("Bitte Geldbetrag eingeben: ");
  scanf("%f", &geldbetrag);
  if (geldbetrag > 0) {geldeinheiten((unsigned long) (geldbetrag*100));}
  return 0;
}
```

Nachlaufen der Funktionsaufrufe

Wird die Rekursion durch eine einzige Variable gesteuert, die sich in einfacher Weise von einem Startwert aus mit jedem erneuten Funktionsselbstaufruf bis zum Basiswert verkleinert, so ergibt sich die *Auflösung der Rekursion* durch *Nachlaufen der Funktionsaufrufe*.

Nachlaufen der Funktionsaufrufe

Wird die Rekursion durch eine einzige Variable gesteuert, die sich in einfacher Weise von einem Startwert aus mit jedem erneuten Funktionsselbstaufruf bis zum Basiswert verkleinert, so ergibt sich die *Auflösung der Rekursion* durch *Nachlaufen der Funktionsaufrufe*.

Beispiel Fakultätsfunktion: $faku(k) = k \cdot faku(k-1)$ und $faku(0) = 1$

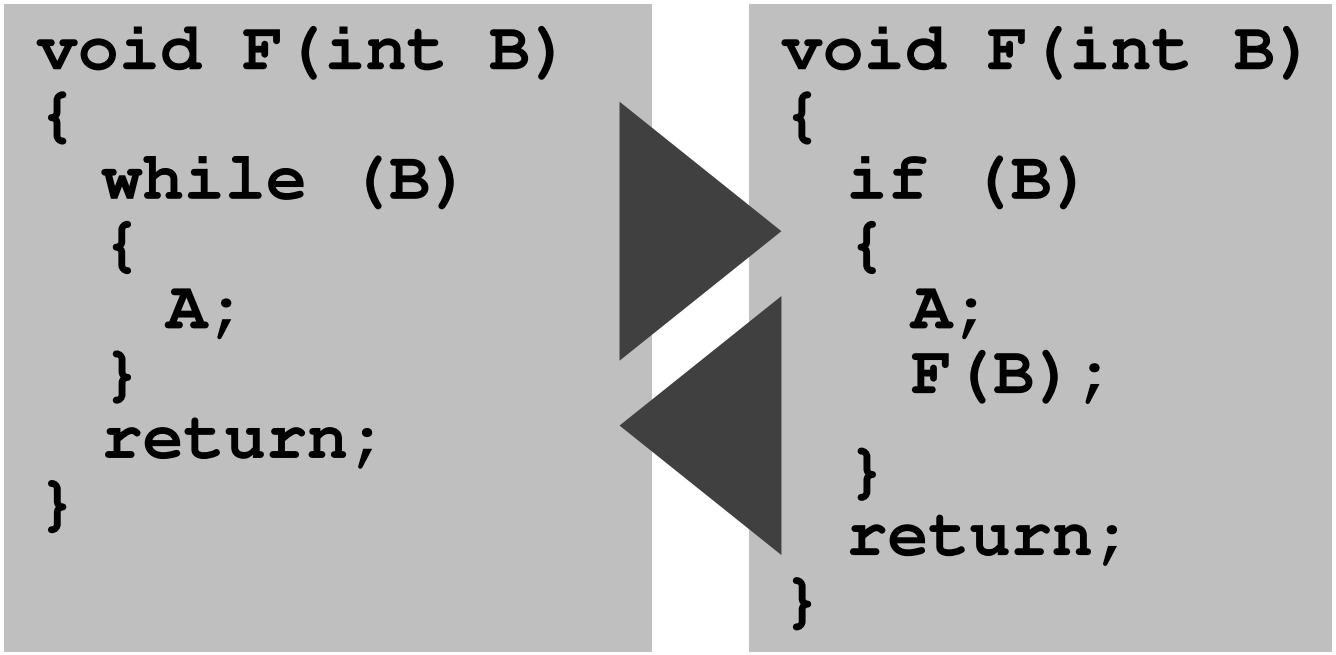
$$\begin{aligned} faku(4) &= 4 \cdot faku(3) \\ &= 4 \cdot 3 \cdot faku(2) \\ &= 4 \cdot 3 \cdot 2 \cdot faku(1) \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \cdot faku(0) \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 \\ &= \prod_{i=1}^{4} i \end{aligned}$$

Allgemein: $faku(k) = \prod_{i=1}^{k} i$ Daraus entsteht unmittelbar eine *iterative Berechnungsmöglichkeit durch Aufmultiplizieren*.

Fakultätsfunktion: Rekursiv und iterativ (faku-iterativ.c)

```c
unsigned long faku_rekursiv(int k)
{
  if (k == 0)
  {
    return 1;
  }
  return k * faku_rekursiv(k-1);
}


unsigned long faku_iterativ(int k)
{
  int i;
  unsigned long p = 1;

  for (i = 1; i <= k; i++)
  {
    p = p * i;
  }
  return p;
}
```

Entsprechung zwischen `while`-Schleife und rekursiver Aufrufkaskade



- **`while`**-Schleife in eine Funktion (hier: **`F`**) eingebettet
- Schleifenbedingung **`B`** agiert als rekursionssteuerndes Argument, in Anweisungsfolge **`A`** kann sich **`B`** verändern
- So jede **`while`**-Schleife in Rekursion transformierbar, andersrum u.U. schwieriger

Rekursion ausreizen

- Die rekursiven Funktionen, die wir bisher betrachtet haben, lassen sich leicht iterativ programmieren. Die Rekursion wurde hier lediglich über eine einzige Variable gesteuert, die in den meisten Beispielen mit jedem Aufruf um eins runtergezählt wird, bis sie den Basiswert erreicht hat. Solche Rekursionen nennt man *primitiv*.

Rekursion ausreizen

- Die rekursiven Funktionen, die wir bisher betrachtet haben, lassen sich leicht iterativ programmieren. Die Rekursion wurde hier lediglich über eine einzige Variable gesteuert, die in den meisten Beispielen mit jedem Aufruf um eins runtergezählt wird, bis sie den Basiswert erreicht hat. Solche Rekursionen nennt man *primitiv*.
- Mit der *Ackermannfunktion* wollen wir uns jetzt eine Funktion anschauen, bei der es schwer ist, eine rein iterative Implementierung zu finden.

Rekursion ausreizen

- Die rekursiven Funktionen, die wir bisher betrachtet haben, lassen sich leicht iterativ programmieren. Die Rekursion wurde hier lediglich über eine einzige Variable gesteuert, die in den meisten Beispielen mit jedem Aufruf um eins runtergezählt wird, bis sie den Basiswert erreicht hat. Solche Rekursionen nennt man *primitiv*.
- Mit der *Ackermannfunktion* wollen wir uns jetzt eine Funktion anschauen, bei der es schwer ist, eine rein iterative Implementierung zu finden.
- Die Ackermannfunktion ist ein Beispiel für eine *besonders schnell wachsende Funktion*, die schon bei kleinen Argumenten äußerst große Funktionswerte annehmen kann.

Rekursion ausreizen

- Die rekursiven Funktionen, die wir bisher betrachtet haben, lassen sich leicht iterativ programmieren. Die Rekursion wurde hier lediglich über eine einzige Variable gesteuert, die in den meisten Beispielen mit jedem Aufruf um eins runtergezählt wird, bis sie den Basiswert erreicht hat. Solche Rekursionen nennt man *primitiv*.
- Mit der *Ackermannfunktion* wollen wir uns jetzt eine Funktion anschauen, bei der es schwer ist, eine rein iterative Implementierung zu finden.
- Die Ackermannfunktion ist ein Beispiel für eine *besonders schnell wachsende Funktion*, die schon bei kleinen Argumenten äußerst große Funktionswerte annehmen kann.
- Die Ackermannfunktion dient in praktischer Anwendung dazu, Compiler und Stackimplementierungen zu testen.

Idee der Ackermannfunktion $ack : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

Zwei Argumente vorsehen: $x, y \in \mathbb{N}$. Eines davon (x) „wandert“ durch die Operationshierarchie, das andere (y) gibt vor, wie oft die Operation angewendet wird.

Argumentbelegung	Operation	Entsprechung
$ack(0, y)$	Inkrementieren	$y + 1$
$ack(1, y)$	Addieren	$y + 2$
$ack(2, y)$	Multiplizieren	$2 \cdot y + 3$
$ack(3, y)$	Potenzieren	$8 \cdot 2^y - 3$
$ack(4, y)$	Hyperpotenzieren	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{y+3 \text{ mal}} - 3$
$ack(5, y)$	...	...

$ack(4, 2) = 2^{65536} - 3$ ist eine Zahl mit 19729 Dezimalstellen.

Rekursive Definition der Ackermannfunktion

$$ack : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

$$\begin{aligned} ack(0, y) &= y + 1 \\ ack(x + 1, 0) &= ack(x, 1) \\ ack(x + 1, y + 1) &= ack(x, ack(x + 1, y)) \end{aligned}$$

Rekursive Definition der Ackermannfunktion

$ack : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$

$$
\begin{aligned}
ack(0, y) &= y + 1 \\
ack(x + 1, 0) &= ack(x, 1) \\
ack(x + 1, y + 1) &= ack(x, ack(x + 1, y))
\end{aligned}
$$

$x \backslash y$	0	1	2	3	4	...
0	1	2	3	4	5	...
1	2	3	4	5	6	...
2	3	5	7	9	11	...
3	5	13	29	61	125	...
4	13	65533	$2^{65536} - 3$	$ack(3, 2^{65536} - 3)$	$ack(3, ack(4, 3))$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Die Ackermannfunktion ist für alle $x, y \in \mathbb{N}$ definiert.

Rekursive Implementierung in C (ack.c)

Ersetzungen: $m = x + 1$, $n = y$

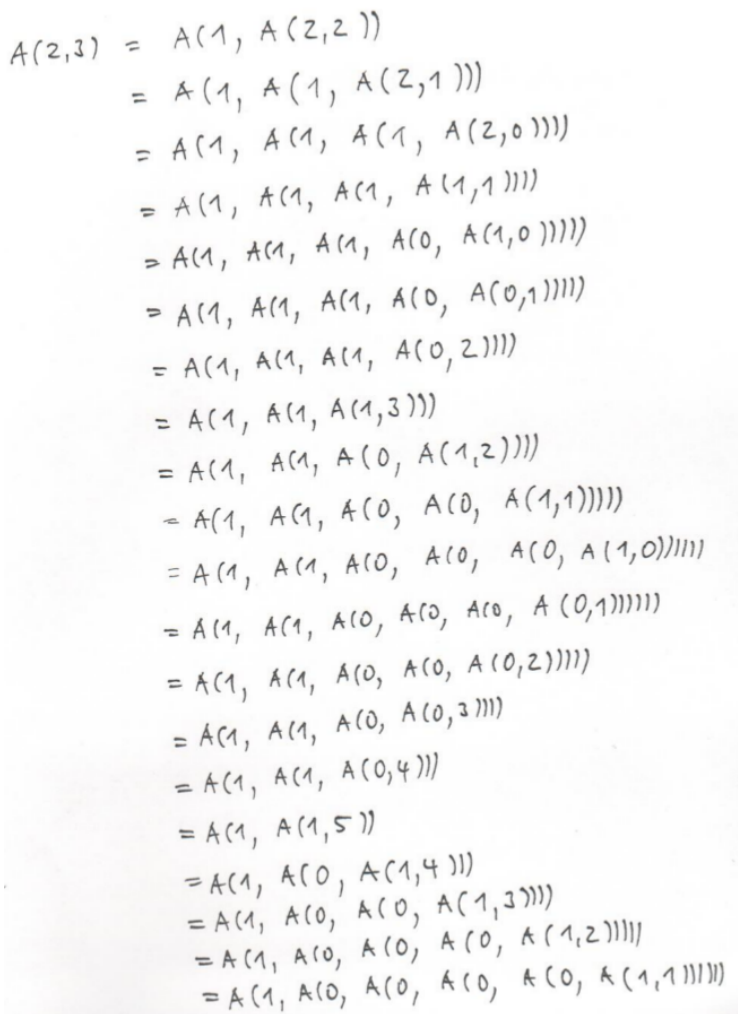
$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

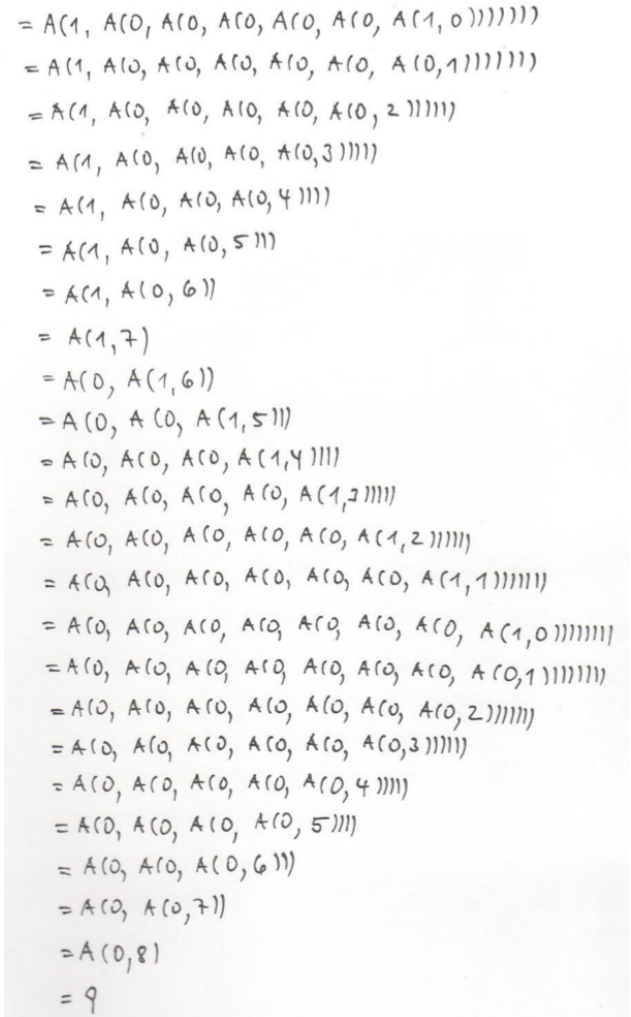
```c
#include <stdio.h>

int ack(int m, int n)
{
  if (m == 0) {return n + 1;}
  if (n == 0) {return ack(m - 1, 1);}
  return ack(m - 1, ack(m, n - 1));
}


int main(void)
{
  int m, n;
  for (m = 0; m <= 4; m++)
  {
    for (n = 0; n < 6 - m; n++)
    {
      printf("ack(%d, %d) = %d\n", m, n, ack(m, n));
    }
  }
  return 0;
}
```

```
ack(0, 0) = 1
ack(0, 1) = 2
ack(0, 2) = 3
ack(0, 3) = 4
ack(0, 4) = 5
ack(0, 5) = 6
ack(1, 0) = 2
ack(1, 1) = 3
ack(1, 2) = 4
ack(1, 3) = 5
ack(1, 4) = 6
ack(2, 0) = 3
ack(2, 1) = 5
ack(2, 2) = 7
ack(2, 3) = 9
ack(3, 0) = 5
ack(3, 1) = 13
ack(3, 2) = 29
ack(4, 0) = 13
ack(4, 1) = 65533
```

Funktionsaufrufstack kann enorme Tiefe erreichen

Aufrufschachtelung zur Berechnung von $ack(2,3) = 9$. Abkürzend: A statt ack

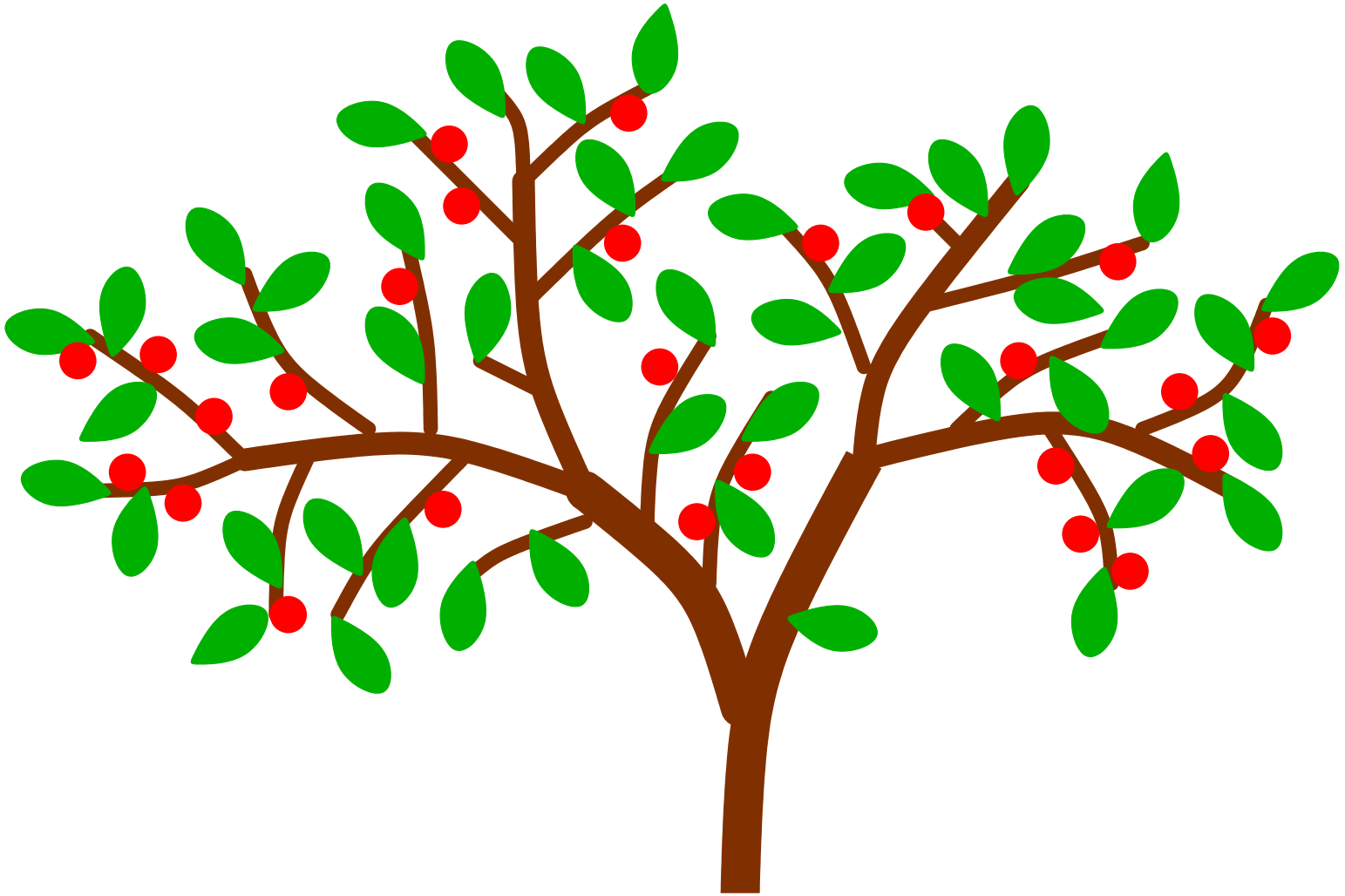




Iterative Implementierung über ein *Feld* (Array) dynamisch programmierbar

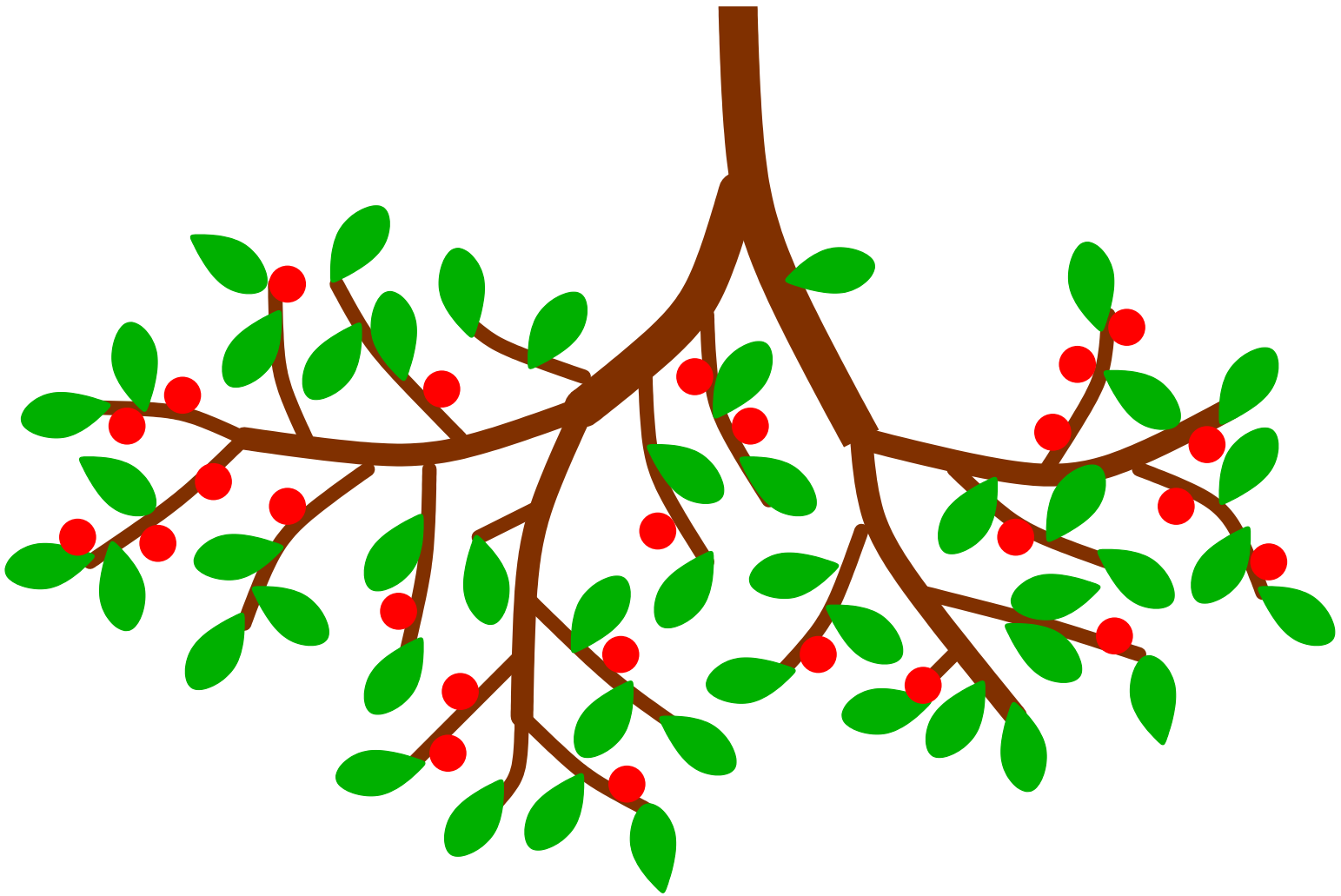
Nochmal unsere Einstiegsfrage

Wieviele Laubblätter und Früchte hängen an diesem Baum?



Bäume des Informatikers wachsen nach unten . . .

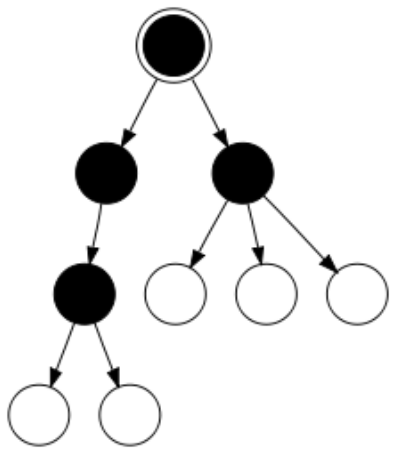
weil er den Baum von der Wurzel aus erschließt



Begriff „Baum“ in der Informatik

In der Informatik versteht man unter einem **Baum** eine netzwerkartige Struktur (*Graph*) aus *Knoten* (Verzweigungs- oder Endpunkte) und *Kanten* (Astabschnitte), die keinen Kantenzyklus enthält.

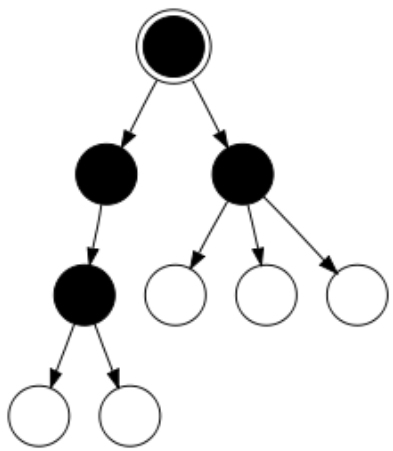
- Üblicherweise gibt es im Baum einen bestimmten Knoten, der die *Wurzel* darstellt.



Begriff „Baum“ in der Informatik

In der Informatik versteht man unter einem **Baum** eine netzwerkartige Struktur (*Graph*) aus *Knoten* (Verzweigungs- oder Endpunkte) und *Kanten* (Astabschnitte), die keinen Kantenzyklus enthält.

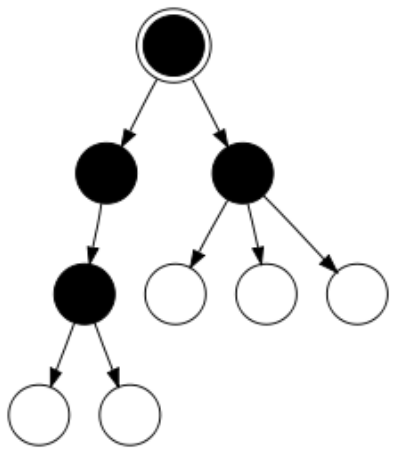
- Üblicherweise gibt es im Baum einen bestimmten Knoten, der die *Wurzel* darstellt.
- Die Baumdurchlaufung („Traversierung“) beginnt in der Regel an der Wurzel über die Kanten zu den nachfolgenden Knoten.



Begriff „Baum“ in der Informatik

In der Informatik versteht man unter einem **Baum** eine netzwerkartige Struktur (*Graph*) aus *Knoten* (Verzweigungs- oder Endpunkte) und *Kanten* (Astabschnitte), die keinen Kantenzyklus enthält.

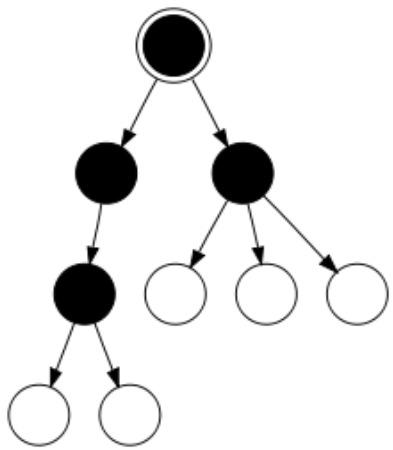
- Üblicherweise gibt es im Baum einen bestimmten Knoten, der die *Wurzel* darstellt.
- Die Baumdurchlaufung („Traversierung“) beginnt in der Regel an der Wurzel über die Kanten zu den nachfolgenden Knoten.
- Knoten ohne Nachfolger heißen *Blätter*.



Begriff „Baum“ in der Informatik

In der Informatik versteht man unter einem **Baum** eine netzwerkartige Struktur (*Graph*) aus *Knoten* (Verzweigungs- oder Endpunkte) und *Kanten* (Astabschnitte), die keinen Kantenzyklus enthält.

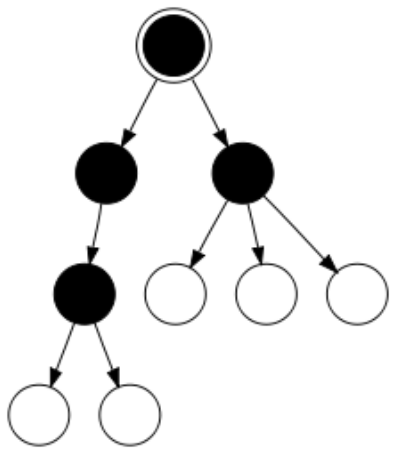
- Üblicherweise gibt es im Baum einen bestimmten Knoten, der die *Wurzel* darstellt.
- Die Baumdurchlaufung („Traversierung“) beginnt in der Regel an der Wurzel über die Kanten zu den nachfolgenden Knoten.
- Knoten ohne Nachfolger heißen *Blätter*.
- Im *binären Baum* hat jeder Knoten höchstens zwei Nachfolger.



Begriff „Baum“ in der Informatik

In der Informatik versteht man unter einem **Baum** eine netzwerkartige Struktur (*Graph*) aus *Knoten* (Verzweigungs- oder Endpunkte) und *Kanten* (Astabschnitte), die keinen Kantenzyklus enthält.

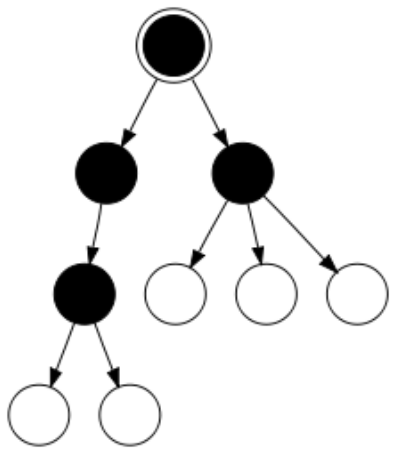
- Üblicherweise gibt es im Baum einen bestimmten Knoten, der die *Wurzel* darstellt.
- Die Baumdurchlaufung („Traversierung“) beginnt in der Regel an der Wurzel über die Kanten zu den nachfolgenden Knoten.
- Knoten ohne Nachfolger heißen *Blätter*.
- Im *binären Baum* hat jeder Knoten höchstens zwei Nachfolger.
- In den Knoten eines Baumes werden üblicherweise *Daten* abgelegt.



Begriff „Baum“ in der Informatik

In der Informatik versteht man unter einem **Baum** eine netzwerkartige Struktur (*Graph*) aus *Knoten* (Verzweigungs- oder Endpunkte) und *Kanten* (Astabschnitte), die keinen Kantenzyklus enthält.

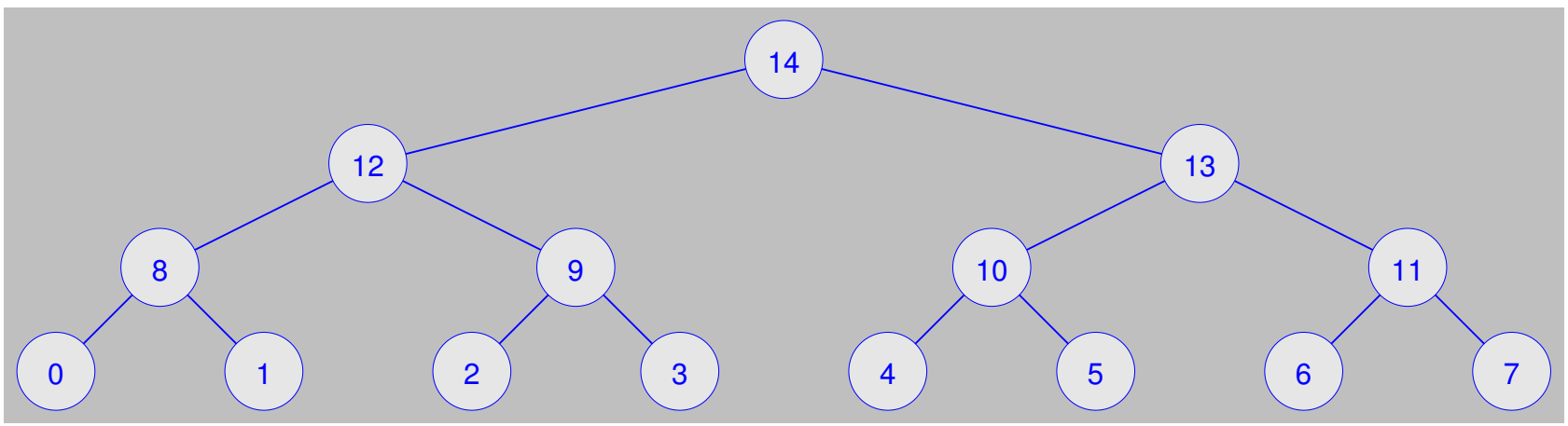
- Üblicherweise gibt es im Baum einen bestimmten Knoten, der die *Wurzel* darstellt.
- Die Baumdurchlaufung („Traversierung“) beginnt in der Regel an der Wurzel über die Kanten zu den nachfolgenden Knoten.
- Knoten ohne Nachfolger heißen *Blätter*.
- Im *binären Baum* hat jeder Knoten höchstens zwei Nachfolger.
- In den Knoten eines Baumes werden üblicherweise *Daten* abgelegt.
- *Hierarchische Strukturen* lassen sich auf diese Weise vorteilhaft modellieren. Beispiel: Verzeichnis- und Dateistruktur auf einer Festplatte.



Knotennummerierung in vollständigem Binärbaum

- Vollständiger Binärbaum:
  Jeder Knoten, der kein Blatt ist, hat genau zwei Nachfolger
- Wähle eine natürliche Zahl $k \geq 1$ entsprechend der Baumgröße
- Der Wurzelknoten hat die Nummer: $m = 2^k - 2$
- Knoten lückenlos ebenenweise von links nach rechts durchnummeriert

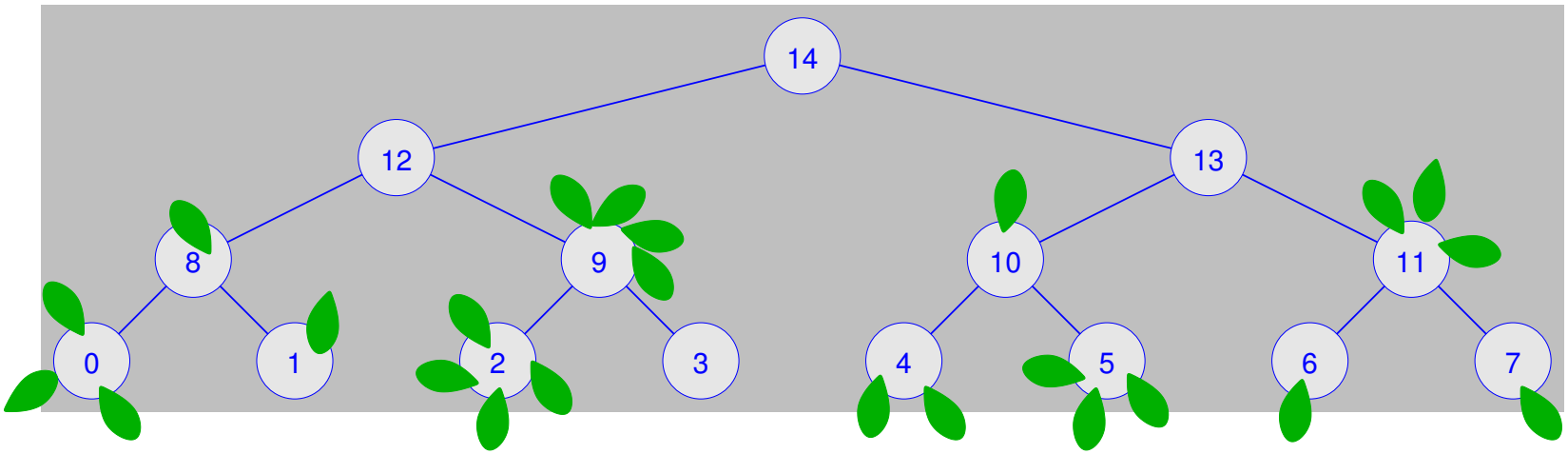
Beispiel mit $k = 4$ und $m = 14$



Laub mittels Baumdurchlaufung zählen

- Knotenfunktion *knoten* liefert zu jeder Knotennummer $0, \ldots, m$ die Anzahl Laubblätter
- Knotenfunktion verkörpert die Daten, die in den Knoten abgelegt werden
- Knotenfunktion eleganter auch als Feld implementierbar

```
int knoten(int i)
{
  switch(i)
  {
    case 0: return 3;
    case 1: return 1;
    case 2: return 4;
    case 3: return 0;
    case 4: return 2;
    case 5: return 3;
    case 6: return 1;
    case 7: return 1;
    case 8: return 1;
    case 9: return 4;
    case 10: return 1;
    case 11: return 3;
    case 12: return 0;
    case 13: return 0;
    case 14: return 0;
  }
  return 0;
}
```



Laub mittels Baumdurchlaufung zählen (bb-summe.c)

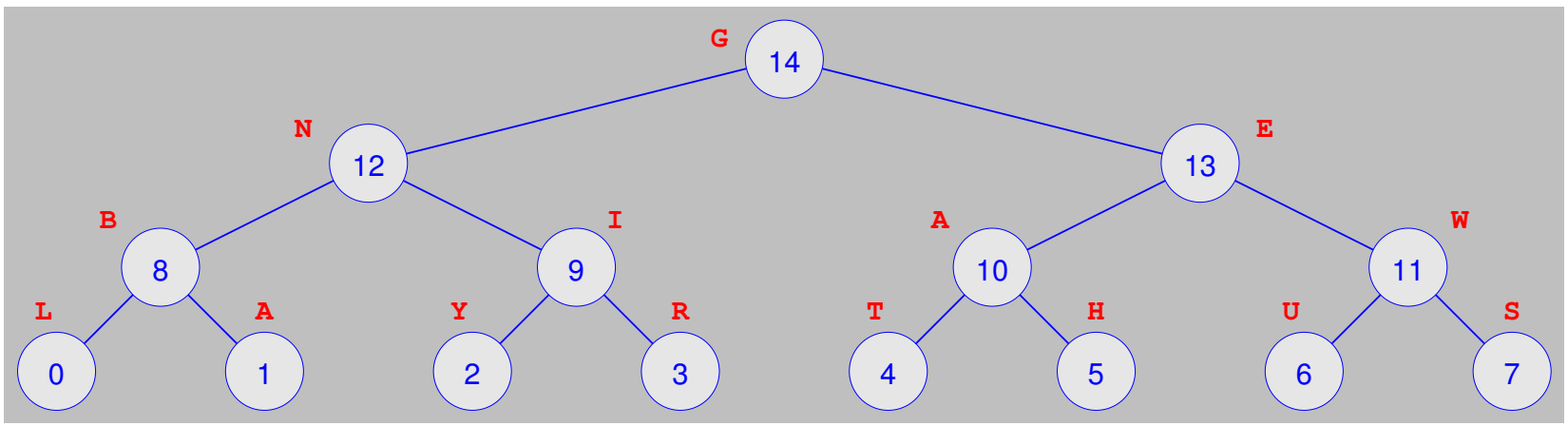
- Rekursiv definierte Funktion *summe* durchläuft den Baum von der Wurzel an.
- Nehmen wir einen beliebigen Knoten n mit $0 \leq n \leq m$.
- Sein *linker Nachfolger* hat die Nummer $(2 \cdot n) \mathrm{mod}(m+2)$.
- Sein *rechter Nachfolger* hat die Nummer $(2 \cdot n) \mathrm{mod}(m+2) + 1$.
- Ist seine Nummer $n \leq \frac{m}{2}$, so handelt es sich um einen *Blattknoten*.

```
int summe(int n, int m)
{
  if (n <= m/2) {return knoten(n);}
  return summe((2*n) % (m+2), m) + summe((2*n) % (m+2) + 1, m) + knoten(n);
}


int main(void)
{
  int k = 4;
  int m = (int) pow(2,k) - 2;
  printf("Anzahl Blaetter: %d\n", summe(m, m));
  return 0;
}
```

postorder-Baumdurchlaufung

- Neue Knotenfunktion *knoten* liefert zu jeder Knotennummer $0, \ldots, m$ einen Buchstaben
- Knotenfunktion verkörpert die Daten, die in den Knoten abgelegt werden
- Knotenfunktion eleganter auch als Feld implementierbar

```
char knoten(int i)
{
  switch(i)
  {
    case 0: return 'L';
    case 1: return 'A';
    case 2: return 'Y';
    case 3: return 'R';
    case 4: return 'T';
    case 5: return 'H';
    case 6: return 'U';
    case 7: return 'S';
    case 8: return 'B';
    case 9: return 'I';
    case 10: return 'A';
    case 11: return 'W';
    case 12: return 'N';
    case 13: return 'E';
    case 14: return 'G';
  }
  return 0;
}
```



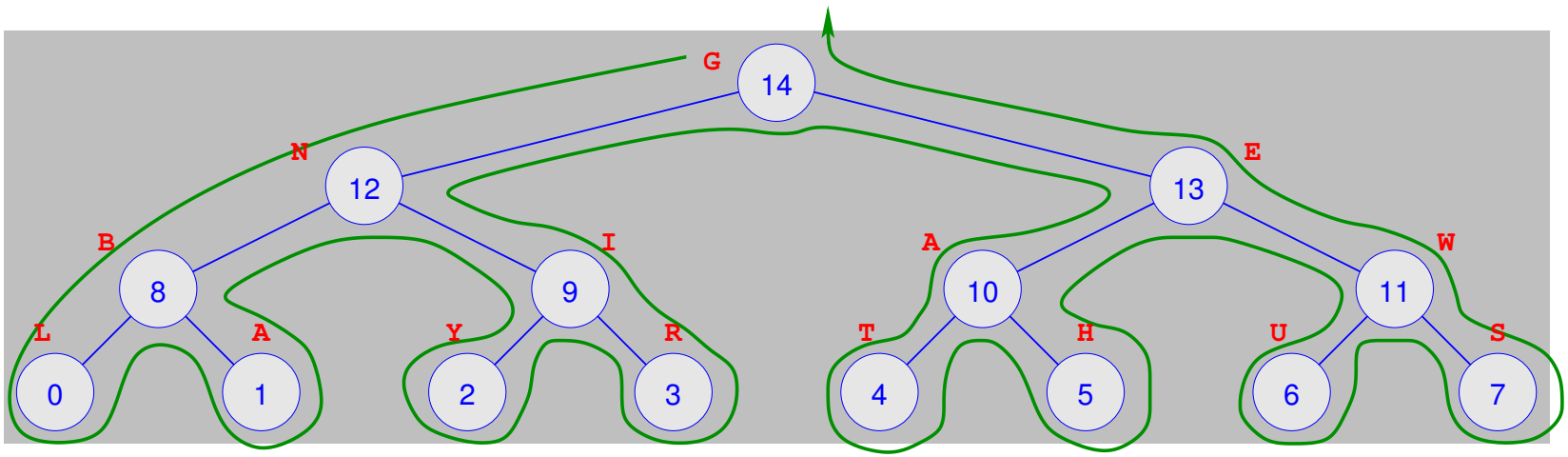
postorder-Baumdurchlaufung (binbaum-postorder.c)

```
void durchlauf(int n, int m)
{
  if (n > m/2) {durchlauf((2*n) % (m+2), m);}      //gehe zum linken Nachfolger
  if (n > m/2) {durchlauf((2*n) % (m+2) + 1, m);} //gehe zum rechten Nachfolger
  printf("%c ", knoten(n));                        //Gib Knoteninhalt aus
  return;
}


int main(void)
{
  int k = 4;
  int m = (int) pow(2,k) - 2;
  durchlauf(m, m);
  printf("\n\n");
  return 0;
}
```

Der rekursiv programmierte C-Quelltext ist *äußerst kompakt*, obwohl das Programm eine durchaus komplizierte Aufgabe löst. Ein semantisch äquivalenter rein iterativer Quelltext wäre länger und schwerer nachzuvollziehen.

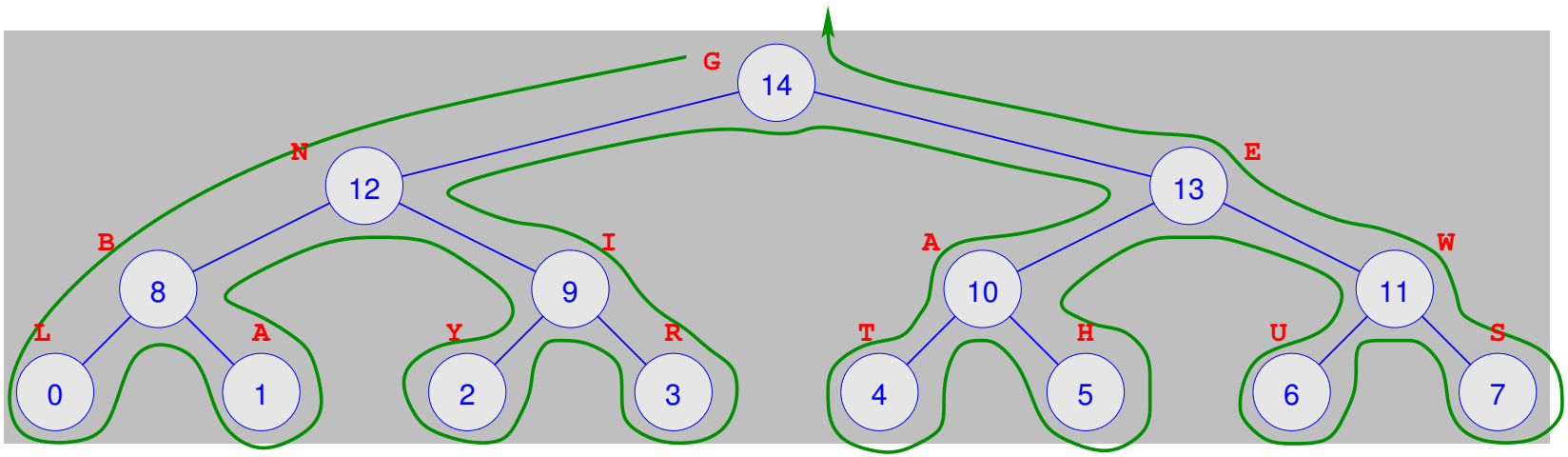
postorder-Baumdurchlaufung (binbaum-postorder.c)

- Durchlauf startet an der Wurzel und verzweigt erst nach links, dann nach rechts
- Sobald ein Knoten *letztmalig* passiert wird, wird sein Inhalt ausgegeben
- Diese Durchlaufungsstrategie heißt *postorder* und ist Grundlage zahlreicher Algorithmen, z.B. Formelparsing



postorder-Baumdurchlaufung (binbaum-postorder.c)

- Durchlauf startet an der Wurzel und verzweigt erst nach links, dann nach rechts
- Sobald ein Knoten *letztmalig* passiert wird, wird sein Inhalt ausgegeben
- Diese Durchlaufungsstrategie heißt *postorder* und ist Grundlage zahlreicher Algorithmen, z.B. Formelparsing




```
L A B Y R I N T H A U S W E G
```