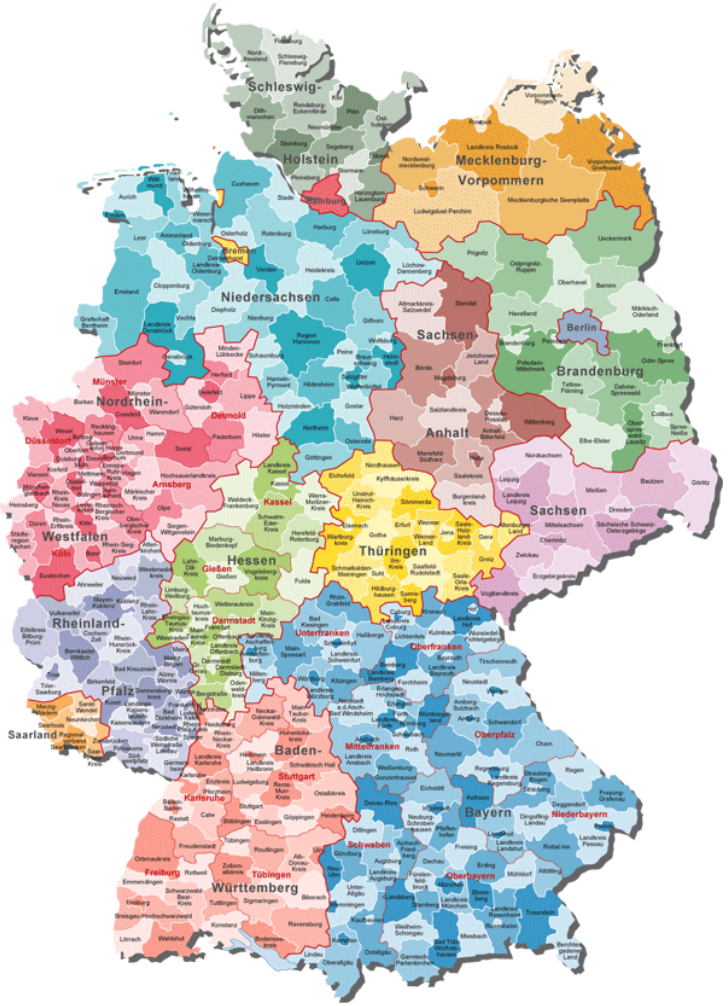
REKURSION

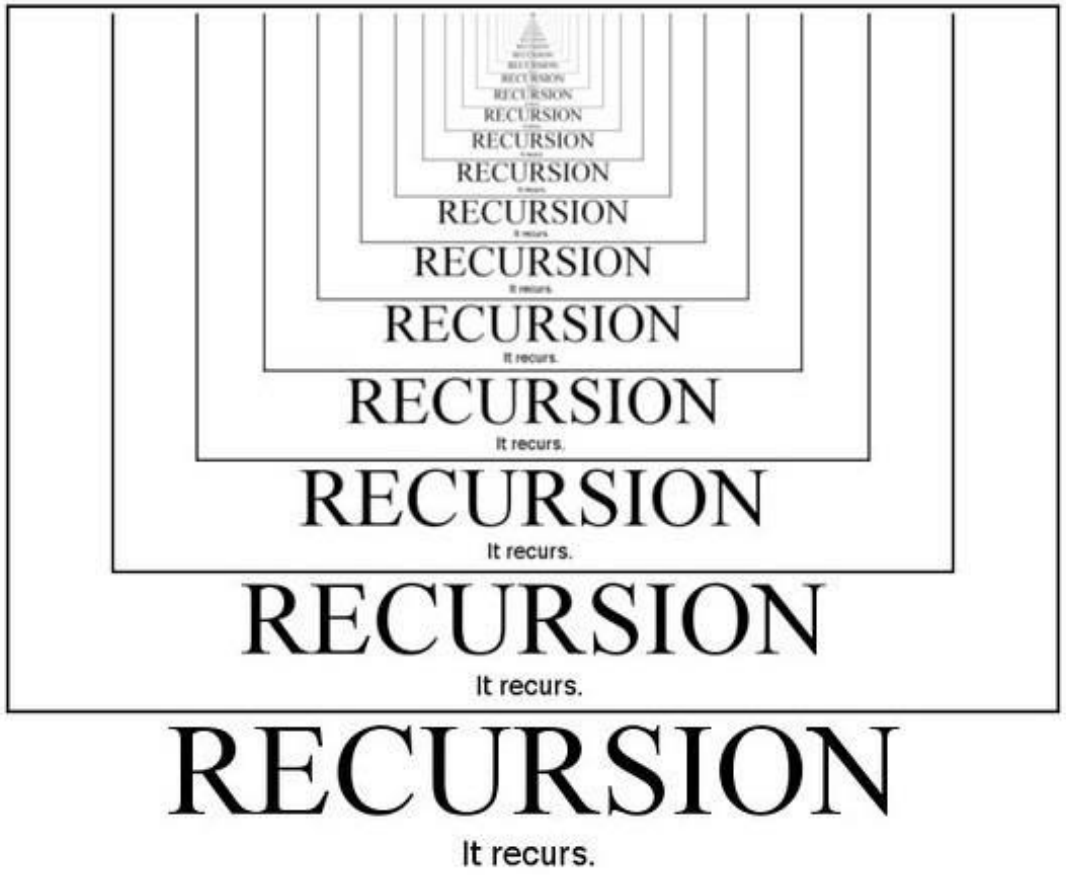
Vincent Uhlmann
IT-Akademie Dr. Heuer GmbH

WAS IST REKURSION?

- Zerlegung komplexer Probleme in kleinere, leicht lösbare Teilprobleme
- Anwendung des "Teile und Herrsche"-Prinzips auf Algorithmen und Funktionen in der Informatik
- Identisches Vorgehen, nur der "Bereich" wird kleiner
- Ein Algorithmus ist rekursiv, wenn in seiner Beschreibung derselbe Algorithmus wieder aufgerufen wird
- In funktionalen Sprachen (Lisp, ML, Haskell etc.) der einzige Weg Schleifen, also Iteration, abzubilden



WAS IST REKURSION?



WAS IST REKURSION?

DER AUFBAU VON REKURSIVEN FUNKTIONEN

- Ein beliebtes Beispiel ist die Berechnung der Fakultät.

$$n! = \begin{cases} 1 & , n = 0 \\ (n-1)! * n & , n > 0 \end{cases}$$

⟸ Einfacher Fall / Abbruch

⟸ Komplexer Fall / Rekursion

- Einfacher Fall:
  - 0! ist 1, d.h. die Fakultät von 0 ist 1
- Komplexer Fall, z.B. 5!
  - 5! ist 4! * 5, d.h. ich muss die Fakultät von 4 berechnen, usw

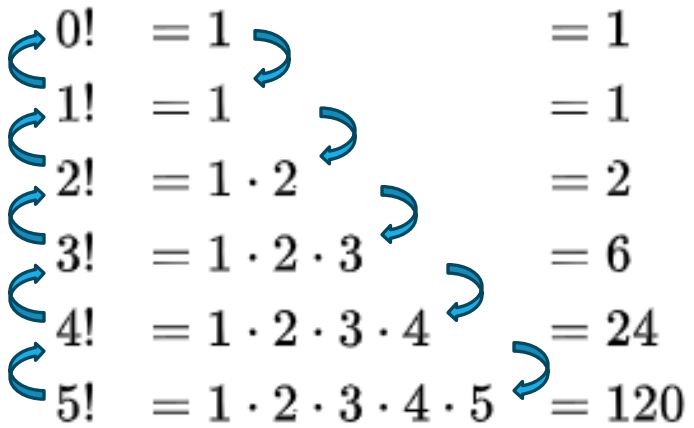
DER AUFBAU VON REKURSIVEN FUNKTIONEN

- Eine rekursive Funktion muss immer eine Abbruchbedingung enthalten! Diese muss immer vor dem Rekursionsaufruf stehen!
- In dem rekursiven Aufruf muss eine Zerlegung oder Vereinfachung des Problems erfolgen!

```
static int Factorial(int n)
{
    if (n == 0)
        return 1;

    return Factorial(n - 1) * n;
}
```

Einfacher Fall / Abbruch

Komplexer Fall / Rekursion



TYPEN VON REKURSIVEN FUNKTIONEN

- Es gibt verschiedene Arten der Rekursion, die sich durch die Art der Position und Anzahl der rekursiven Aufrufe unterscheiden

- Dies hat immer einen Einfluss auf dem Speicherverbrauch (Call-Stack) und die Effizienz der durchgeführten Berechnung.

TYPEN VON REKURSIVEN FUNKTIONEN

- Bei der Linearenrekursion tritt in jedem Zweig die Rekursion höchstens einmal auf. Bei der Auswertung der Funktion ergibt sich eine lineare Folge von rekursiven Aufrufen

```
static int LinearRecursive(int n)
{
    if (n == 0)
        return 1;

    return LinearRecursive(n - 1) * n;
}
```

$$n! = \begin{cases} 1 & , n = 0 \\ (n-1)! * n & , n > 0 \end{cases}$$

Fakultät

TYPEN VON REKURSIVEN FUNKTIONEN

- Bei der Endrekursion wird der Rückgabewert des rekursiven Aufrufs direkt an den Aufrufer zurückgegeben, und es erfolgt keine weitere Verarbeitung nach dem rekursiven Aufruf
- Endrekursionen können oft effizient durch Compiler-Optimierungen umgesetzt werden

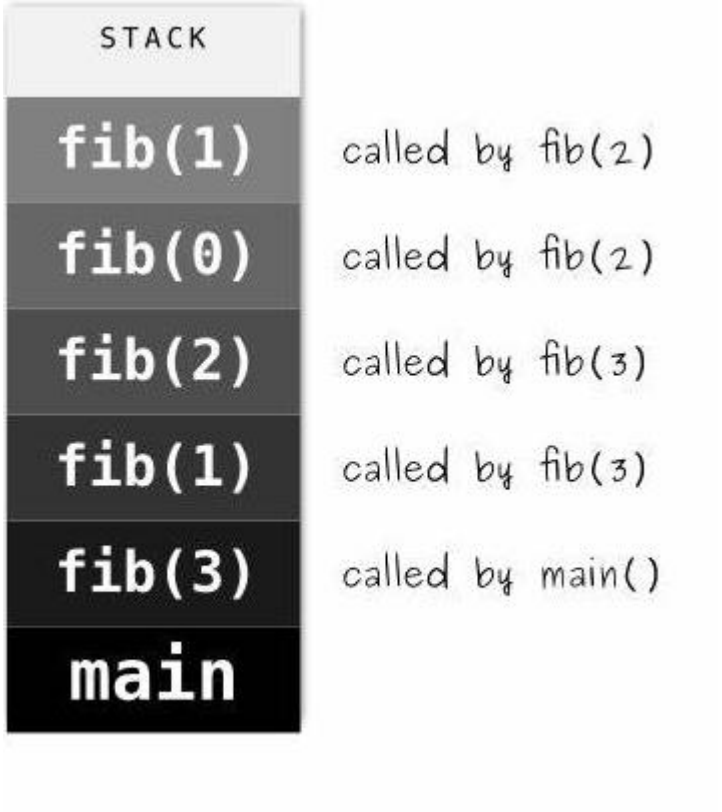
```
static int TailRecursive(int n, int result = 1)
{
    if (n == 0)
        return result;

    return TailRecursive(n - 1, n * result);
}
```

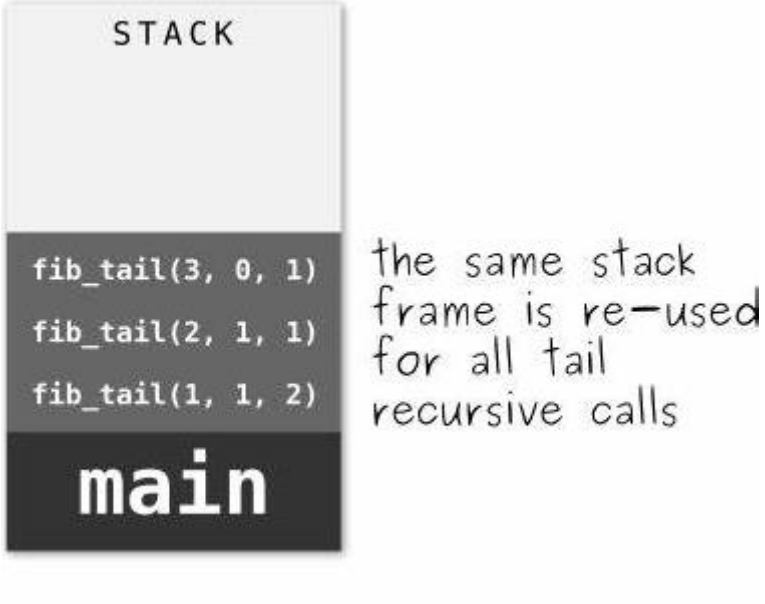
$$n! = \begin{cases} 1 & , n = 0 \\ (n-1)! * n & , n > 0 \end{cases}$$

Fakultät

TYPEN VON REKURSIVEN FUNKTIONEN

- Endrekursionen können oft effizient durch Compiler-Optimierungen umgesetzt werden





TYPEN VON REKURSIVEN FUNKTIONEN

- Baumrekursion tritt auf, wenn die Funktion in mindestens einem ihrer Zweige mehrfach aufgerufen wird
- Der Funktionsaufruf kann mehrere Kinder in einem "Baum" von rekursiven Aufrufen haben

```
static int TreeRecursive(int n, int k)
{
    if (k == 0 || k == n)
        return 1;

    return TreeRecursive(n - 1, k - 1) + TreeRecursive(n - 1, k);
}
```

$$\binom{n}{k} = \begin{cases} 1 & k = 0,\ k = n, \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n. \end{cases}$$

Binomialkoeffizient

TYPEN VON REKURSIVEN FUNKTIONEN

- Verschachtelte Rekursion bezieht sich auf den Fall, in dem der rekursive Aufruf der Funktion selbst wiederum einen rekursiven Aufruf als Argument enthält
- Der Aufruf der Funktion erfolgt verschachtelt oder geschachtelt

```
static int NestedRecursive(int n, int m)
{
    if (n == 0)
        return m + 1;

    if (n != 0 && m == 0)
        return NestedRecursive(n - 1, 1);

    return NestedRecursive(n - 1, NestedRecursive(n, m - 1));
}
```

$$f(n,m) = \begin{cases} m+1 & n = 0, \\ f(n-1,1) & n \neq 0, m = 0, \\ f(n-1, f(n, m-1)) & n \neq 0, m \neq 0. \end{cases}$$

Ackermann-Funktion

TYPEN VON REKURSIVEN FUNKTIONEN

- Verschränkte Rekursion bezieht sich darauf, dass der rekursive Aufruf der Funktion in irgendeiner Weise wechselseitig oder indirekt erfolgt.
- Es gibt eine Art von indirekter Abhängigkeit zwischen den rekursiven Aufrufen.

```
static bool Even(int n)
{
    if (n == 0)
        return true;

    return Odd(n - 1);
}
```

```
static bool Odd(int n)
{
    if (n == 0)
        return false;

    return Even(n - 1);
}
```

$$\text{even}(n) = \begin{cases} \text{true} & n = 0, \\ \text{odd}(n-1) & n > 0. \end{cases}$$

$$\text{odd}(n) = \begin{cases} \text{false} & n = 0, \\ \text{even}(n-1) & n > 0. \end{cases}$$