Rekursion

>> Eine Implementierung eines Algorithmus bezeichnen wir als besonders **elegant**, wenn sie ein Problem auf eine interessante und leicht visualisierbare Art löst.

Die Technik der **Rekursion** ist eine sehr verbreitete Methode, um eine solche elegante Lösung zu implementieren.

Die Definition einer rekursiven Funktion ist, dass sie sich im Rahmen ihrer Ausführung selbst aufruft.

Fakultäts-funktion

Die Fakultät $n!$ ist für alle positiven ganzen Zahlen definiert.

$n!$ ist das Produkt aller positiven ganzen Zahlen kleiner oder gleich n.

Beim Programmieren nennen wir die mathematische Funktion $n!$ oft `fact(n)`.

Rekursion bei der Fakultätsfunktion

```
fact(1) = 1

fact(2) = 2 * 1

fact(3) = 3 * 2 * 1

fact(4) = 4 * 3 * 2 * 1

fact(5) = 5 * 4 * 3 * 2 * 1
```

Rekursion bei der Fakultätsfunktion

```
fact(1) = 1

fact(2) = 2 * fact(1)

fact(3) = 3 * 2 * 1

fact(4) = 4 * 3 * 2 * 1

fact(5) = 5 * 4 * 3 * 2 * 1
```

Rekursion bei der Fakultätsfunktion

```
fact(1) = 1

fact(2) = 2 * fact(1)

fact(3) = 3 * fact(2)

fact(4) = 4 * 3 * 2 * 1

fact(5) = 5 * 4 * 3 * 2 * 1
```

Rekursion bei der Fakultätsfunktion

```
fact(1) = 1

fact(2) = 2 * fact(1)

fact(3) = 3 * fact(2)

fact(4) = 4 * fact(3)

fact(5) = 5 * 4 * 3 * 2 * 1
```

Rekursion bei der Fakultätsfunktion

```
fact(1) = 1

fact(2) = 2 * fact(1)

fact(3) = 3 * fact(2)

fact(4) = 4 * fact(3)

fact(5) = 5 * fact(4)
```

Rekursion bei der Fakultätsfunktion

```
fact(1) = 1

fact(2) = 2 * fact(1)

fact(3) = 3 * fact(2)

fact(4) = 4 * fact(3)

fact(5) = 5 * fact(4)

fact(n) = n * fact(n-1)
```

Zwei Fälle

Jede rekursive Funktion implementiert zwei mögliche Fälle:

1. den **Basisfall**, der die Rekursion beendet;

2. den **rekursiven Fall**, in dem die eigentliche Rekursion stattfindet.

Rekursion bei der Fakultätsfunktion

```
fact(1) = 1
```

```
int fact(int n)
{
    // Basisfall
    if (n == 1)
    {
        return 1;
    }

    // Rekursiver Fall ...
}
```

Rekursion bei der Fakultätsfunktion

```
fact(n) = n * fact(n-1)
```

```
int fact(int n)
{
    // Basisfall
    if (n == 1)
    {
        return 1;
    }
    // Rekursiver Fall
    else
    {
        return n * fact(n-1);
    }
}
```

```
int fact(int n)
{
    if (n == 1)
        return 1;
    else
        return n * fact(n-1);
}
```

Iterative Variante

```
fact(n) = n * fact(n-1)
```

```
int fact2(int n)
{
    int product = 1;
    while(n > 0)
    {
        product *= n;
        n--;
    }
    return product;
}
```

```
int fact(int n)
{
    if (n == 1)
        return 1;
    else
        return n * fact(n-1);
}
```

Besondere Fälle

Meistens, aber nicht immer, ersetzen rekursive Funktionen Schleifen in nicht-rekursiven Funktionen.

Es ist auch möglich, **mehr als einen Basisfall** oder **mehr als einen Rekursionsfall** zu haben.

Beispiele:
Fibonacci-Zahlen, Collatz-Vermutung

Fibonacci-Folge

Die Fibonacci-Folge ist wie folgt definiert:

Das erste Element ist 0.

Das zweite Element ist 1.

Das n-te Element ist die Summe
der Elemente ($n-1$) und ($n-2$).

Die Collatz-Vermutung

Die Collatz-Vermutung gilt für positive ganze Zahlen. Es wird vermutet, dass man immer „zurück zur 1“ kommt, wenn man folgende Schritte befolgt:

Wenn $n = 1$, gib 1 zurück.

Sonst:

  wenn n gerade ist, gib $n / 2$ zurück;

  wenn n ungerade ist, gib $3n + 1$ zurück.

Wir könnten eine rekursive Funktion c(n) schreiben, die berechnet, wie viele Schritte man braucht, um zur 1 zu gelangen, wenn man bei n *beginnt.*

c(*n*): Anzahl der Schritte bis zur 1

n	c(n)	Schritte
1	0	1
2	1	2 1
3	7	3 10 5 16 8 4 2 1
4	2	4 2 1
5	5	5 16 8 4 2 1
6	8	6 3 10 5 16 8 4 2 1
7	16	7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
8	3	8 4 2 1
15	17	15 46 23 70 ... 8 4 2 1
27	111	27 82 41 124 ... 8 4 2 1
50	24	50 25 76 38 ... 8 4 2 1

Implementieren Sie c(n).

Lösung

```
int collatz(int n)
{
    // base case
    if (n == 1)
        return 0;
    // even numbers
    else if ((n % 2) == 0)
        return 1 + collatz(n/2);
    // odd numbers
    else
        return 1 + collatz(3*n + 1);
}
```

EXTRAS IN 3 MINUTEN

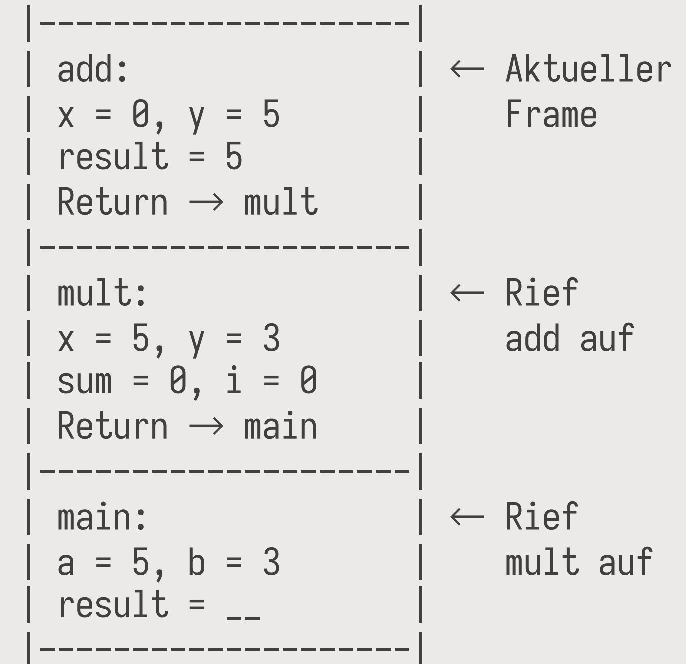
FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

Entscheidungsproblem zur Laufzeit

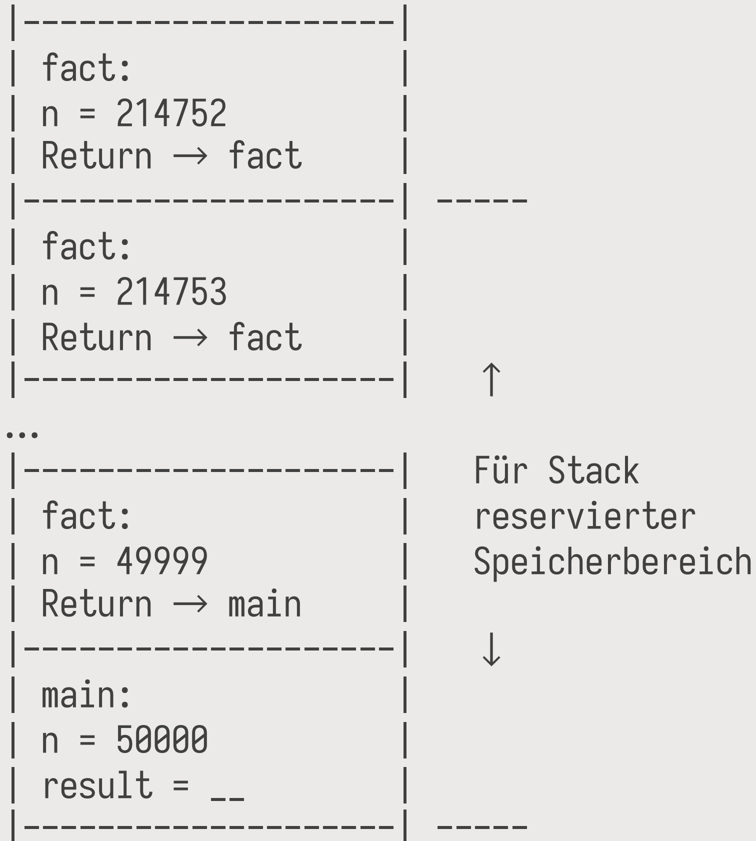
```c
int add_two(int x) {
    return add(x, 2);
}

int add(int x, int y) {
    return result = x + y; // Wo geht es weiter: in add_two oder mult?
}

int mult(int x, int y) {
    int sum = 0;
    for(int i = 0; i < y; i++) {
        sum = add(sum, x);
    }
    return sum;
}

int main() {
    int a = 5, b = 3;
    int result = mult(a, b);
    printf("%d", result);
    return 0;
}
```

```c
int add_two(int x) {
    return add(x, 2);
}

int add(int x, int y) {
    return result = x + y; // ← hier sind wir
}

int mult(int x, int y) {
    int sum = 0;
    for(int i = 0; i < y; i++) {
        sum = add(sum, x);
    }
    return sum;
}

int main() {
    int a = 5, b = 3;
    int result = mult(a, b);
    printf("%d", result);
    return 0;
}
```

Lösung: Call Stack im Speicher



Stack Overflow



Was berechnet dieses Programm?

```c
#include <stdio.h>


int waysToClimbStairs(int n);


int main(void)
{
    int n = 4;
    printf("Result for n = %d is %d\n", n, waysToClimbStairs(n));
    return 0;
}


int waysToClimbStairs(int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return waysToClimbStairs(n - 1) +
               waysToClimbStairs(n - 2);
}
```

Rekursion:

Eine Funktion ruft sich selbst mit kleineren Teilproblemen auf.

Der rekursive Fall reduziert das Problem auf einen einfacheren Fall.

Jede rekursive Funktion benötigt mindestens einen Basisfall.

Rekursion ist oft eleganter als iterative Lösungen.

Der Basisfall beendet die Rekursion und liefert ein direktes Ergebnis.

Achtung: Rekursionstiefe durch Stackgröße begrenzt.