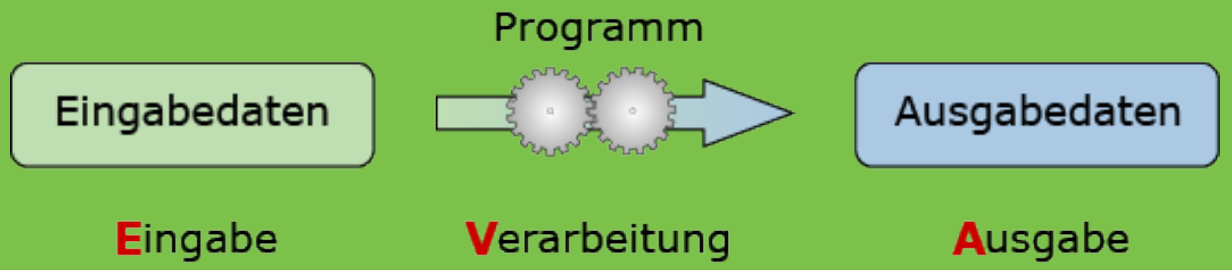
Hochschule Fulda
University of Applied Sciences

Programmierung 1 – Rekursion



Yvonne Jung

Wiederholung: Iteration

Hochschule Fulda
University of Applied Sciences

- Anweisungen im Schleifenrumpf werden wiederholt ausgeführt
  - Schleife wird mittels Abbruchbedingung beendet
    - Sonst Endlosschleife…
- *Iterative* Lösung der Fakultätsfunktion
  - Entsprechend Formel: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{k=1}^{n} k \qquad 0! = 1$
  - C-Code (mit Schleife):

```
unsigned fakultaet(unsigned n) {
    unsigned res = 1;
    for (unsigned i=2; i<=n; i++)
        res *= i;
    return res;
}
```

Fakultätsberechnung

Hochschule Fulda
University of Applied Sciences

- Statt iterativ alternativ *rekursiv* möglich
  - Rekursive Definition der Fakultät:

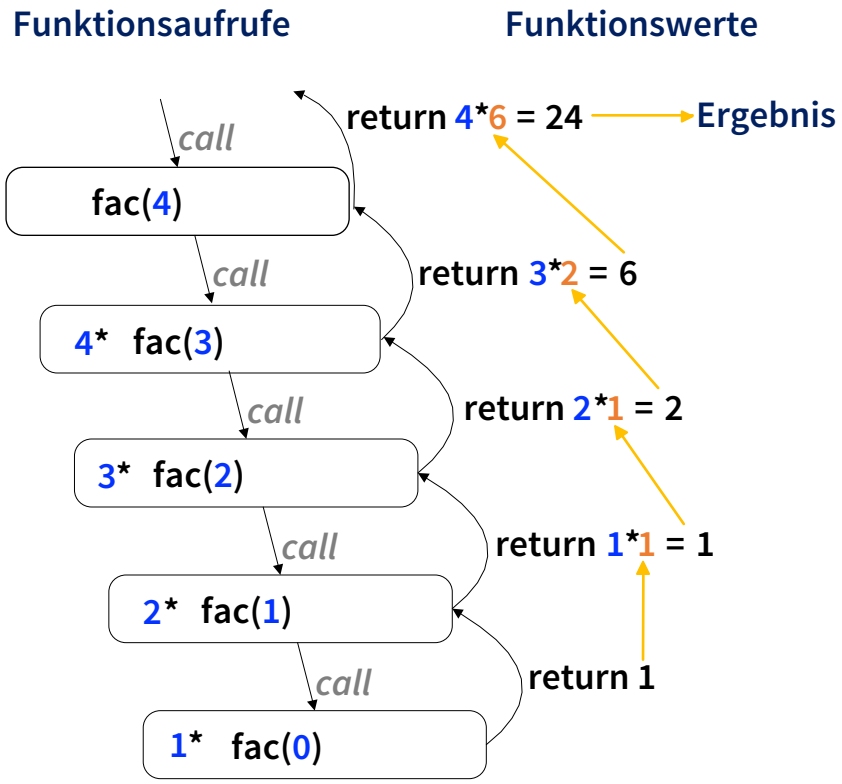
$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

- C-Code:

```
unsigned fac(unsigned n) {
    if (n == 0)
        return 1;
    return n * fac(n - 1);
}
```

  - Kasten für jeden rekursiven Aufruf
    - Mit Pfeil vom Aufrufer zur aufgerufenen Funktion
    - Pfeil zurück zum Aufrufer zeigt je Rückgabewert

Gesucht: 4!



Rekursive Funktionen

Hochschule Fulda
University of Applied Sciences

- Eine Funktion wie $fac(n)$, die sich wieder selbst aufruft, heißt *rekursiv*
  - Ein solcher Aufruf kann auch indirekt über eine weitere Funktion erfolgen
- Durch Rekursion wird ein Problem "auf sich selbst" zurückgeführt
  - Funktioniert nur, wenn Problem durch diese Rückführung *einfacher* wird
    - Zur Berechnung von $fac(n)$ werden Ergebnisse für *kleinere* Eingaben verwendet, d.h. $fac(n-1)$
  - …und wenn es *Abbruchbedingung* für Rekursion gibt
- Eine rekursive Funktion besteht aus…
  - einem oder mehreren Basisfällen
    - Dienen als *Abbruchbedingung* und sind meist einfach implementierbar, wie z.B. für $fac(0)$
  - dem rekursiven Fall

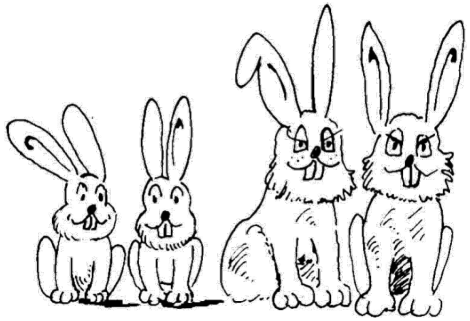
```
unsigned fac(unsigned n) {
    if (n == 0)
        return 1;
    return n * fac(n - 1);
}
```

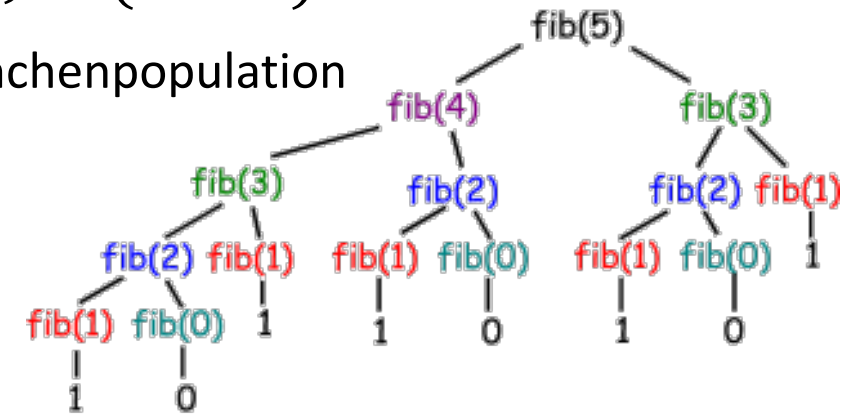
Fibonacci-Folge

- Die Fibonacci-Zahlen sind rekursiv wie folgt definiert:

$$fib(n) = \begin{cases} 0 & (n = 0) \\ 1 & (n = 1) \\ fib(n-1) + fib(n-2) & (n \geq 2) \end{cases}$$

  - L. Fibonacci beschrieb damit Wachstum einer Kaninchenpopulation





- Tabellarisch:

n	0	1	2	3	4	5	6	7	8	9	…
fib(n)	0	1	1	2	3	5	8	13	21	34	…

Implementierung

- Rekursiv

```
unsigned fibonacciRek(unsigned n) {
    if (n <= 1)
        return n;
    return fibonacciRek(n-1) + fibonacciRek(n-2);
}
```

- Iterativ

```
unsigned fibonacciIter(unsigned n) {
    unsigned f0 = 0, f1 = 1, res = n;
    for (int i=2; i<=n; i++) {
        res = f0 + f1;
        f0 = f1;
        f1 = res;
    }
    return res;
}
```

Rekursion

Hochschule Fulda
University of Applied Sciences

- Auf zwei Punkte achten
  - **Rekursiver Fall / Rekursionsschritt:**
    Argumente des rekursiven Aufrufs müssen Aufgabe darstellen, die *einfacher* zu lösen ist als die, die Aufrufer übergeben wurde → Argumente müssen "kleiner" werden
  - **Basisfall / Terminierung:**
    Bei jedem Aufruf prüfen, ob Aufgabe *ohne* erneute Rekursion gelöst werden kann
- Rekursion ist Programmiermethode
  - Dabei wird Problem gelöst, indem es auf einfachere Instanz zurückgeführt wird
  - Was einfacher bzw. kleiner bedeutet, hängt von den verwendeten Datentypen ab
    - Integer: kleinere Zahl, Basisfall oft 0 oder 1
    - Array: nur Teil des Arrays wird noch verwendet, Basisfall oft leeres Array
    - String: kürzerer String, Basisfall oft leerer String

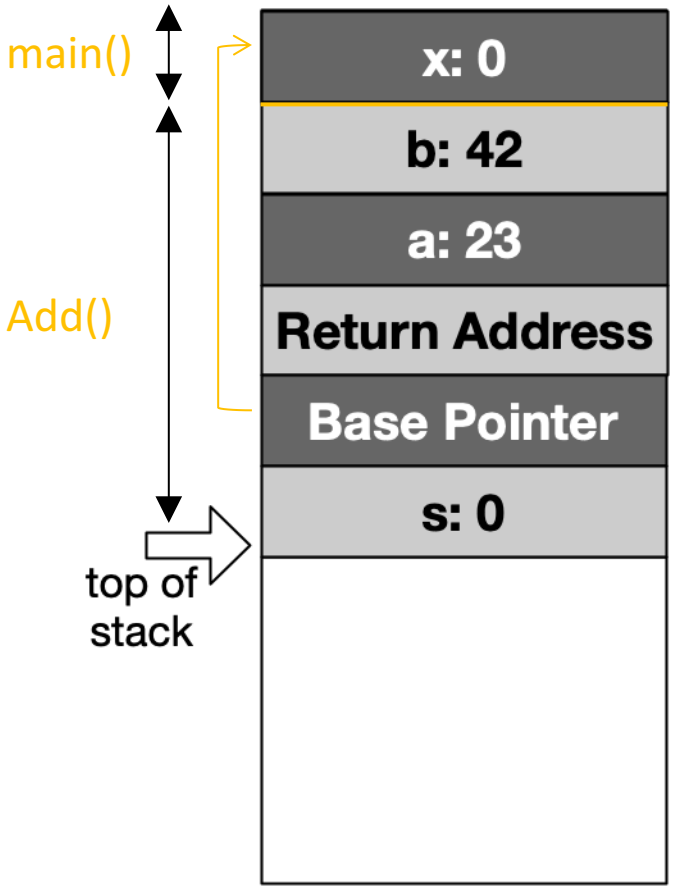
Einschub: Call Stack (1)

Hochschule Fulda
University of Applied Sciences

- Funktion braucht Platz, um lokale Variablen zu speichern
  - Zur Erinnerung: Bereich heißt *Stack*
    - Besteht aus Speicherplätzen mit numerischer Adresse
    - Variablen ordnen diesen Adressen Namen zu
  - Alle Daten und Variablen eines Funktionsaufrufs befinden sich in *Stack Frame*
- Bei Funktionsaufruf wird neuer Stack Frame angelegt
  - Es entsteht ein Aufrufstapel (Call Stack)
    - In Hochsprachen automatisch vom Compiler erledigt
  - Parameter (sowie danach Rückgabewerte) werden kopiert
  - Nach Beendigung wird Stack Frame an OS zurückgegeben
    - Achtung, Stack Size endlich, kann bei vielen Rekursionen zu Stack Overflow führen

Einschub: Call Stack (2)

Hochschule Fulda
University of Applied Sciences




```
int Add(int a, int b)
{
    int s = 0;
    s = a + b;

    return s;
}

int main()
{
    int x = 0;

    x = Add(23,42);

    return 0;
}
```

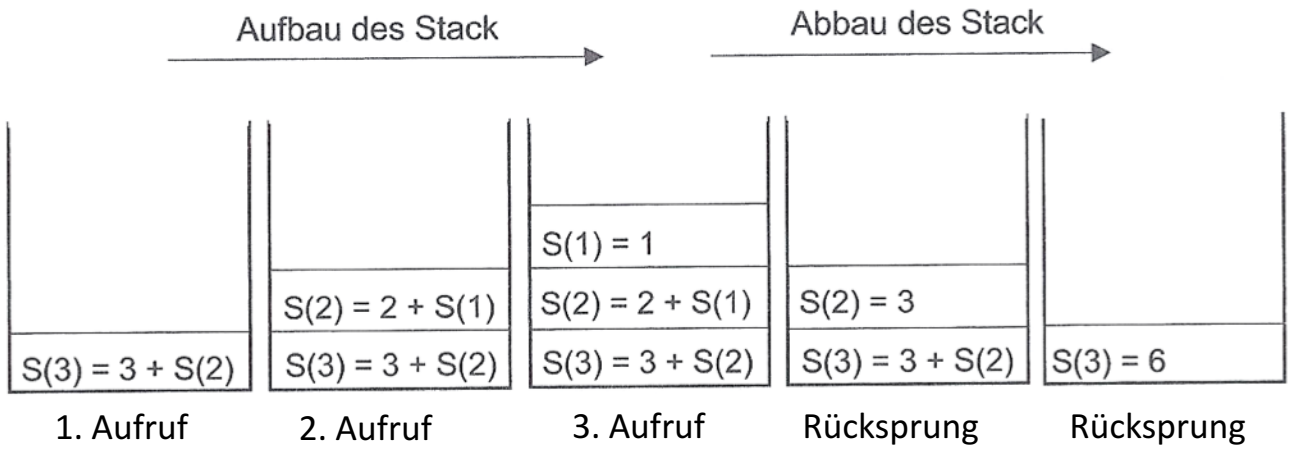
- Auf Call Stack werden noch nicht zurückgekehrte Funktionsaufrufe verwaltet
  - Im Beispiel wurde einfache Funktion *Add()* aufgerufen
- Achtung: bei nicht abbrechender Rekursion erfolgt Stack Overflow

Rekursive Summenfunktion

Hochschule Fulda
University of Applied Sciences

- Summenformel: $S(n) = \sum_{i=0}^{n} i = {}^{n(n+1)}/_{2}$
- Rekursive Formulierung:
- $S(n) = \begin{cases} n + S(n-1) & n > 1 \\ 1 & n = 1 \end{cases}$
- Berechnungsschritte für $n = 3$

```
unsigned sum(unsigned n) {
    if (n == 0)                    Basisfall
        return 0;
    return n + sum(n-1);           Rekursionsfall
}
```

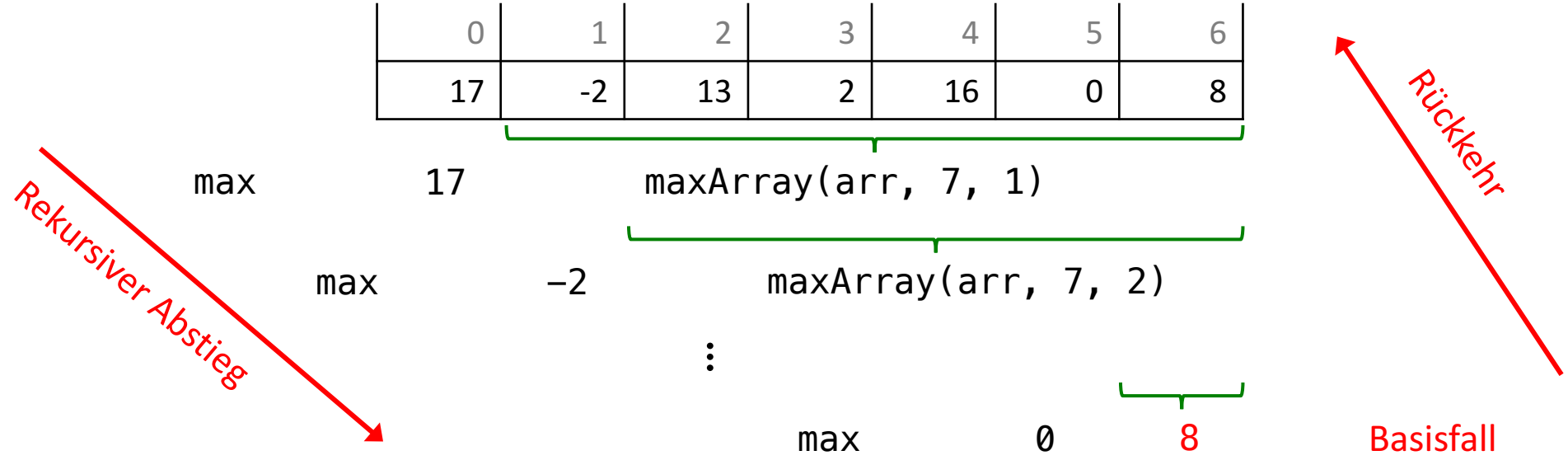


S(3) = 3 + S(2)
S(2) = 2 + S(1)
S(1) = 1
S(2) = 2 + 1 = 3
S(3) = 3 + 3 = 6

Rekursion über Arrays

Hochschule Fulda
University of Applied Sciences

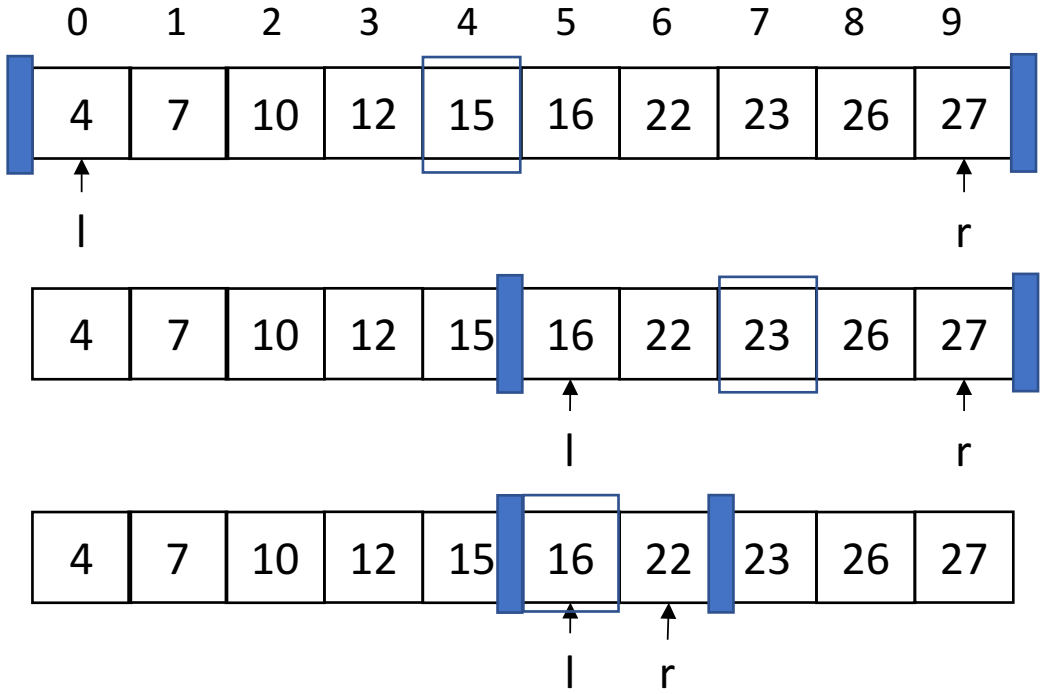
- Rekursiver Abstieg erfolgt über kleiner werdende Feldlänge
  - Basisfall: betrachtete Restlänge 0 oder 1
  - Implementierung über je angepasste Start-/End-Indizes
- Beispiel: Suche maximales Element in Feld (ab Position *maxAb*)
  - `int maxArray(int arr[], int n, int maxAb);`



Binäre Suche (1)

Hochschule Fulda
University of Applied Sciences

- Beispiel 1: Suche nach Zahl 16 in sortiertem Feld



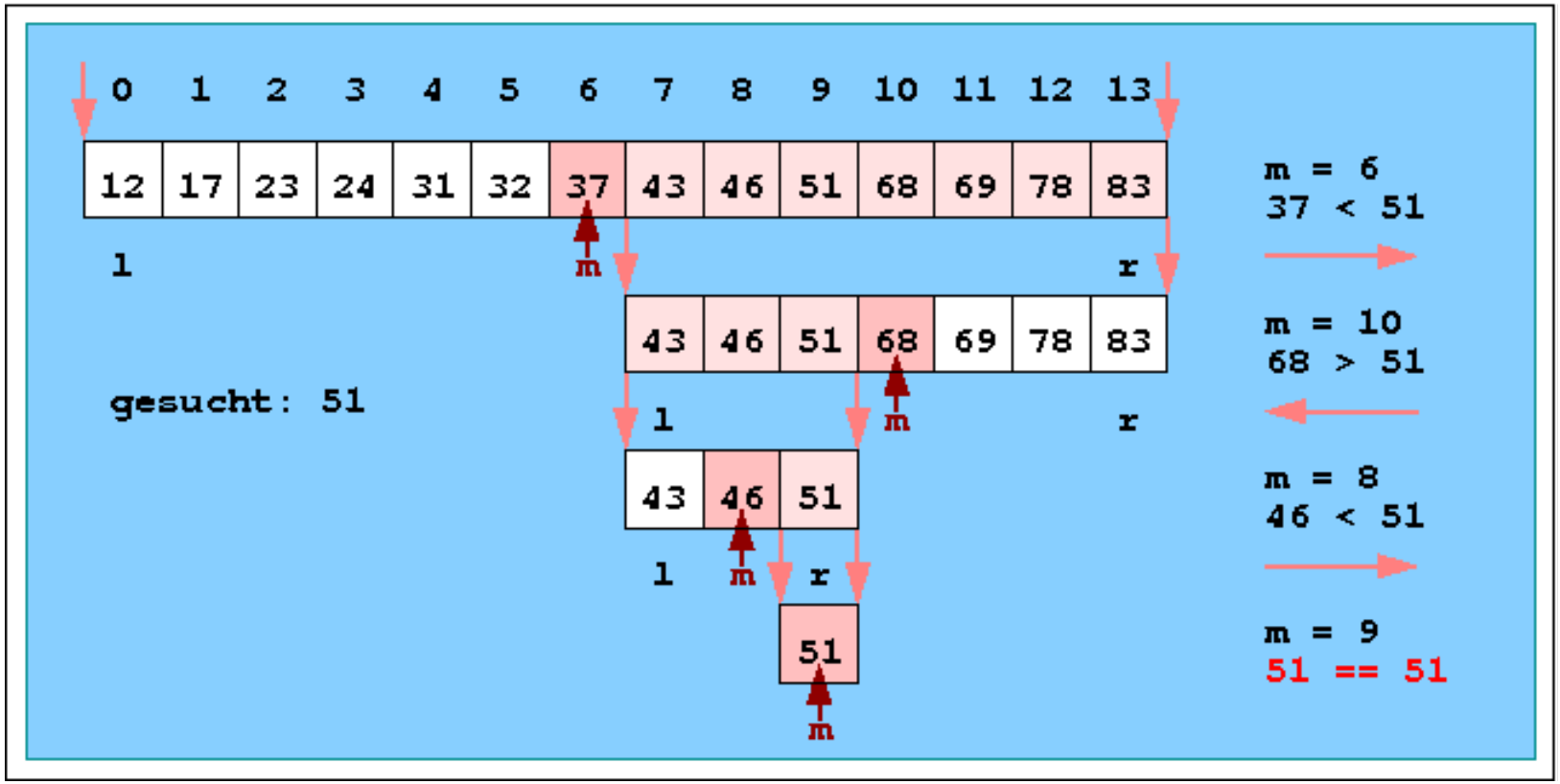

```
int l = 0, r = 9,
    m = (l + r) / 2;
```

Bei Rekursion über Arrays nutzt man i.d.R. Hilfsparameter, um Suchbereich anzugeben

Binäre Suche (2)

Hochschule Fulda
University of Applied Sciences

- Beispiel 2: Suche nach Zahl 51 in sortiertem Feld



Binäre Suche (3)

Hochschule Fulda
University of Applied Sciences

- Algorithmus
  1. Wähle mittleren Eintrag des *sortierten* Feldes der Länge n
  2. Falls dies noch nicht gesuchtes Element ist, dann prüfe, ob Element in erster oder zweiter Hälfte liegt
  3. Zurück zu Punkt 1 für die Hälfte, in der sich Element befinden müsste
- Anmerkungen
  - In jedem Schritt wird Suchbereich (Anzahl zu prüfender Elemente) halbiert
  - Für ein Array der Länge n benötigt man höchstens $\log_2 n$ Tests
  - Relativ einfach, iterativ zu implementieren, wenn man rekursive Lösung kennt
    - Iterativ und rekursiv haben hier gleiche Performance
    - Das gilt bekanntlich aber oft nicht (vgl. z.B. Fibonacci)

Implementierung

Hochschule Fulda
University of Applied Sciences

```
int binarySearch(int arr[], int left, int right, int elem) {
    if (left > right)
        return -1;

    int mid = (left + right) / 2;
    if (arr[mid] == elem)
        return mid;

    if (elem < arr[mid])
        return binarySearch(arr, left, mid - 1, elem);
    else
        return binarySearch(arr, mid + 1, right, elem);
}
```

- Arbeitet nach Prinzip „Divide and Conquer“ (Teile und Herrsche)

Probleme und Grenzen

Hochschule Fulda
University of Applied Sciences

- Iteration versus Rekursion
  - Prinzipiell immer beides möglich
  - Rekursion: einfach, aber meist ineffizient
  - Iteration: schwieriger, aber meist effizienter
- Vorsicht
  - Bei manchen rekursiven Funktion ist die Berechnungsdauer nicht abschätzbar
    - Beispiel: Ackermann-Funktion

$$a(0, m) = m + 1 \qquad \forall\ m \geq 0$$
$$a(n, 0) = a(n - 1, 1) \qquad \forall\ n \geq 1$$
$$a(n, m) = a(n - 1, a(n, m - 1)) \qquad \forall\ n, m \geq 1$$

    - Übersteigt selbst bei sehr kleinen Eingabewerten schnell alle Berechnungsmöglichkeiten

Vielen Dank!

Noch Fragen?