

Rekursion

Andreas M. Hohenauer

Algorithmen und Datenstrukturen

14. September 2025

Definition

Definition (Rekursion)

Unter einer rekursiven Methode versteht man eine Methode die sich selbst aufruft.

- Die Definition gilt auch für *Funktionen* oder *Programme* (statt Methoden).
- Viele Problemstellungen/Algorithmen lassen sich sehr einfach und elegant mittels Rekursion formulieren.
- Abbruchbedingung: muss vorhanden sein um die rekursiven Aufrufe zu beenden.

Fakultät

In der Mathematik bezeichnet $n!$ die Fakultät, und es gilt

$$n! = n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1$$

für alle $n > 0$ und $0! = 1$.

Rekursive Formulierung der Fakultät:

$$n! = n \cdot (n - 1)! \quad \text{für } n > 0$$

$$0! = 1$$

Direkte Umsetzung in Programmcode:

Algorithmus 1: Rekursive Fakultät

Input: $n \in \mathbb{N}_0$

Output: $n!$

```

1 Function FACTORIAL( $n$ )
2   if  $n \leq 0$  then
3     return 1
4   return  $n \cdot \text{FACTORIAL}(n - 1)$ 
```

Negativbeispiel: Fibonacci-Zahlen

Die Folge der Fibonacci-Zahlen ist für $n \geq 2$ definiert als

$$f_n = f_{n-1} + f_{n-2},$$

weilers gilt $f_0 = f_1 = 1$.

Hierdurch erhalten wir die Folge:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Negativbeispiel: Fibonacci-Zahlen

Die direkte Umsetzung als rekursives Programm ist jedoch sehr unvorteilhaft:

Algorithmus 2: Naive Fibonacci

Input: $n \in \mathbb{N}_0$

Output: n -ten Folgengleid: F_n

```

1 Function FIBREC( $n$ )
2   if  $n \leq 1$  then
3     return 1
4   return FIBREC( $n - 1$ ) + FIBREC( $n - 2$ )

```

Frage

Wo genau liegt das Problem bei der rekursiven Implementierung der Fibonacci-Zahlen?

Negativbeispiel: Fibonacci-Zahlen

Die *iterative* Implementierung ist hier vorteilhaft, da wesentlich weniger Berechnungen ausgeführt werden:

Algorithmus 3: Iterativer Fibonacci

Input: $n \in \mathbb{N}_0$

Output: F_n

```

1 Function FIBITER( $n$ )
2   if  $n < 0$  then
3     return Fehler
4   if  $n \leq 1$  then
5     return 1
6    $f[0] \leftarrow 1$ 
7    $f[1] \leftarrow 1$ 
8   for  $i \leftarrow 2$  to  $n$  do
9      $f[i] \leftarrow f[i-1] + f[i-2]$ 
10  return  $f[n]$ 

```

Anmerkung: Die Berechnung des n -ten Folgegliedes ist auch ohne Array möglich!

Suche in sortierten Listen

- Aufgabenstellung: Suchen eines Elementes in einer sortierten Liste (Array)
- Naiver Zugang: $O(n)$ Schritte (Erwartungswert $n/2$)
- Mittels binärer Suche: nur $O(\log n)$ Schritte notwendig
- Vorgehensweise: analog zu Suche in Telefonbuch
 - Beliebige Seite aufschlagen
 - Steht Name davor oder danach?
 - Weitere Suche erfolgt nur mehr im verbleibenden Teil

Binäre Suche

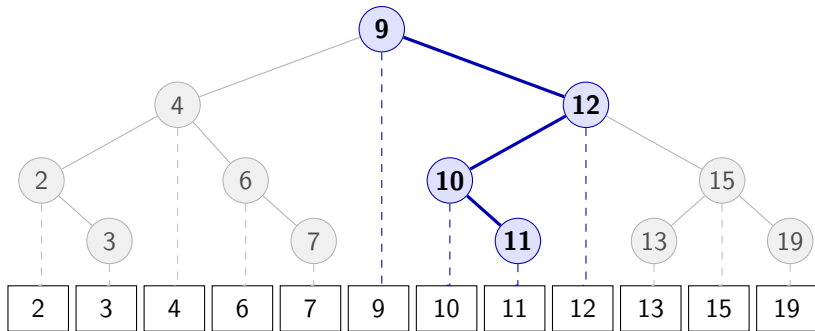


Abbildung: Binäre Suche nach 11 im Array [2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 15, 19]. Im Wurzelknoten wird nach rechts gegangen, da $11 > 9$. Die Mitte der rechten Teilfolge enthält das Element 12. Davon muss nach links weiter gegangen werden, u.s.w.

Binäre Suche

Algorithmus 4: Rekursive Binäre Suche

Input: Array A , linke und rechte Grenze l und r , gesuchtes Element x

```

1 Function BINSUCHE( $A, l, r, x$ )
2   if  $l > r$  then
3     return NOTFOUND
4    $m = l + \lfloor (r - l) / 2 \rfloor$ 
5   if  $A[m] = x$  then
6     return  $m$ 
7   else if  $A[m] < x$  then
8     return BINSUCHE( $A, m + 1, r, x$ )
9   else
10    return BINSUCHE( $A, l, m - 1, x$ )
  
```

Aufruf mit array a und gesuchtem Element e und $\text{BINSUCHE}(a, 0, a.\text{length}-1, e,)$.

Rekursive Binäre Suche

Eine rekursive Variante in Python:

```

1  def binaersuche_rekursiv(werte, gesucht, start, ende):
2      if ende < start:
3          return 'nicht gefunden'
4          # alternativ: return -start # bei (-Returnwert) waere
5          # die richtige Einfuege-Position
6
7      mitte = (start + ende) // 2
8      if wert[e[mitte]] == gesucht:
9          return mitte
10     elif wert[e[mitte]] < gesucht:
11         return binaersuche_rekursiv(werte, gesucht, mitte+1, ende)
12     else:
13         return binaersuche_rekursiv(werte, gesucht, start, mitte-1)
14
15 def binaersuche(werte, gesucht):
16     return binaersuche_rekursiv(werte, gesucht, 0, len(werte)-1)
  
```

Quelle: Wikipedia

Tiefensuche

Bei der Tiefensuche ist eine rekursive Implementierung naheliegend. Diese Variante durchläuft alle vom ersten Aufruf mit Knoten $v \in V(G)$ ($\text{DFS}(v)$) aus erreichbaren Knoten des Graphen G .

Algorithmus 5: Rekursive Tiefensuche

Input: $G = (V, E), v \in V$

```

1 Function DFS( $G, v$ )
2   markiere  $v$  als besucht
3   for alle  $u$  mit  $\{v, u\} \in E$  do
4     if  $u$  nicht besucht then
5       DFS( $G, u$ )
  
```

Die Tiefensuche kann mittels der Datenstruktur *Stack* auch *iterativ* (also ohne Rekursion) implementiert werden.

Tiefensuche

Algorithmus 6: Tiefensuche zur Suche eines Knoten s

Input: $G = (V, E)$, $v \in V$ (aktueller Knoten), s (gesuchter Knoten)

Output: s falls gefunden, sonst NULL

```
1 Function DFS( $G, v, s$ )
2   if  $v = s$  then
3     return  $v$ 
4   markiere  $v$  als besucht;
5   for alle  $u$  mit  $\{v, u\} \in E$  do
6     if  $u$  nicht besucht then
7       if DFS( $G, u, s$ ) ==  $s$  then
8         return  $s$ 
9   return NULL
```

Programmieraufgabe

Gegeben ist ein 2-dimensionales Boolean-Array, befüllt mit *wahr* und *falsch*. Ermitteln Sie mit einem Programm die Größe des größten zusammenhängenden Gebietes mit Wert wahr.

Beispiel: hier wird falsch mit einem '.' und wahr mit einem 'X' dargestellt.

```

...XXX..XX..
.XXX.XXXX...
X.XX..XX..XX
X...XXX..XXX
XXX....XXX..
XX.XX..X..XX
...X..X.XXX.
.....XXXX...
```

In diesem Beispiel ist die Größe 19.

Zusatzaufgabe

Füllen Sie ein zweidimensionales Boolean-Array (Größe 300x300) mit Zufallswerten, wobei mit einer Wahrscheinlichkeit von $1/3$ der Wert wahr, und mit $2/3$ der Wert falsch gesetzt werden soll. Wenden Sie den Algorithmus aus dem vorigen Beispiel an. Ist das Ergebnis plausibel? Was passiert wenn man die Wahrscheinlichkeiten tauscht?

```
1  private static boolean[][] werte = {  
2      {false, false, false, true, true, true, false, false, true, true, false, false},  
3      {false, true, true, true, false, true, true, true, true, false, false, false},  
4      {true, false, true, true, false, false, true, true, false, false, true, true},  
5      {true, false, false, false, true, true, true, false, false, true, true, true},  
6      {true, true, true, false, false, false, false, true, true, true, false, false},  
7      {true, true, false, true, true, false, false, true, false, false, true, true},  
8      {false, false, false, true, false, false, true, false, true, true, true, false},  
9      {false, false, false, false, false, true, true, true, true, false, false, false}  
10 };
```