Rekursion

Andreas M. Hohenauer

Algorithmen und Datenstrukturen

12. 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

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

für alle $n > 0$ und $0! = 1$.

Rekursive Formulierung der Fakultät:

$$\begin{aligned} n! &= n \cdot (n-1)! \qquad \text{für } n > 0 \\ 0! &= 1 \end{aligned}$$

Direkte Umsetzung in Programmcode:

Algorithmus 1: Rekursive Fakultät

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

```
Function FACTORIAL(n)
    if n = 0 then
        return 1
```

Fakultät

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

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

für alle $n > 0$ und $0! = 1$.

Rekursive Formulierung der Fakultät:

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

Direkte Umsetzung in Programmcode:

Algorithmus 2: Rekursive Fakultät

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

```
Function FACTORIAL(n)
    if n = 0 then
        return 1
```

Negativbeispiel: Fibonacci-Zahlen

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

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

weiters gilt $f_0 = f_1 = 1$.

Hierdurch erhalten wir die Folge:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$$

Negativbeispiel: Fibonacci-Zahlen

Die direkte Umsetzung als rekursives Programm ist jedoch sehr unvorteilhaft:

Algorithmus 3: Naive Fibonacci

Input: $n \in \mathbb{N}_0$
Output: n-ten Folgengleid: F_n

```
Function FibRec(n)
    if n ≤ 1 then
        return 1
    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 ausgeführte werden:

Algorithmus 4: Iterativer Fibonacci

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

Function $\textsc{FibIter}(n)$
    if $n < 0$ **then**
        return Fehler
    if $n \leq 1$ **then**
        return 1
    $f[0] \leftarrow 1$
    $f[1] \leftarrow 1$
    for $i \leftarrow 2$ **to** n **do**
        $f[i] \leftarrow f[i-1] + f[i-2]$
    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

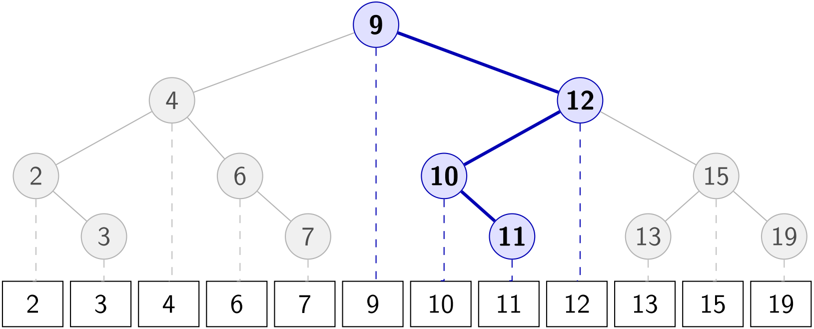

Abbildung: Binäre Suche nach 11 im Array $[2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 15, 19]$. Im Wurzelknoten wird nach rechts gegangen, da $11 > 9$. Die Mitte der rechten Teilfolge enthält das Element 12. Davon muss nach links weiter gegangen werden, u.s.w.

Binäre Suche

Algorithmus 5: Rekursive Binäre Suche

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

Function BinSuche(A, l, r, x)
    if $l > r$ **then**
        return NotFound
    $m = l + \lfloor (r-l)/2 \rfloor$
    if $A[m] = x$ **then**
        return m
    else if $A[m] < x$ **then**
        return BinSuche$(A, m+1, r, x)$
    else
        return BinSuche$(A, l, m-1, x)$

Aufruf mit array a und gesuchtem Element e und BinSuche`(a, 0, a.length-1, e,)`.

Rekursive Binäre Suche

Eine rekursive Variante in Python:

```python
def binaersuche_rekursiv(werte, gesucht, start, ende):
    if ende < start:
        return 'nicht gefunden'
        # alternativ: return -start # bei (-Returnwert) waere
        # die richtige Einfuege-Position

    mitte = (start + ende) // 2
    if werte[mitte] == gesucht:
        return mitte
    elif werte[mitte] < gesucht:
        return binaersuche_rekursiv(werte, gesucht, mitte+1, ende)
    else:
        return binaersuche_rekursiv(werte, gesucht, start, mitte-1)

def binaersuche(werte, gesucht):
    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 $v \in V(G)$ (`DFS(v)`) aus erreichbaren Knoten des Graphen G .

Algorithmus 6: Rekursive Tiefensuche

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

Function $\mathrm{DFS}(G, v)$
  markiere v als besucht
  for *alle* u *mit* $\{v, u\} \in E$ **do**
    if u *nicht besucht* **then**
      $\mathrm{DFS}(G, u)$

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

Tiefensuche

Algorithmus 7: Tiefensuche zur Suche eines Knoten s

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

Output: s falls gefunden, sonst NULL

```
Function DFS(G, v, s)
    if v = s then
        return v
    markiere v als besucht;
    for alle u mit {v, u} ∈ E do
        if u nicht besucht then
            if DFS(G, u, s) == s then
                return s
    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?

```
	private static boolean[][] werte = {
			{false,false,false,true,true,true,false,false,true,true,false,false},
			{false,true,true,true,false,true,true,true,true,false,false,false},
			{true,false,true,true,false,false,true,true,false,false,true,true},
			{true,false,false,false,true,true,true,false,false,true,true,true},
			{true,true,true,false,false,false,false,true,true,true,false,false},
			{true,true,false,true,true,false,false,true,false,false,true,true},
			{false,false,false,true,false,false,true,false,true,true,true,false},
			{false,false,false,false,false,true,true,true,true,false,false,false}
			};
```