

Rekursion

Theorie (Kurzversion)

Sinn und Zweck

Manchmal ist es sinnvoll, anstelle einer Schleife eine Funktion zu schreiben, die sich selber (rekursiv) aufruft. Dann muss innerhalb der Funktion eine Bedingung definiert sein, die für den Abbruch der Rekursion sorgt (**Base Case**) und zu ihrer Auflösung führt.

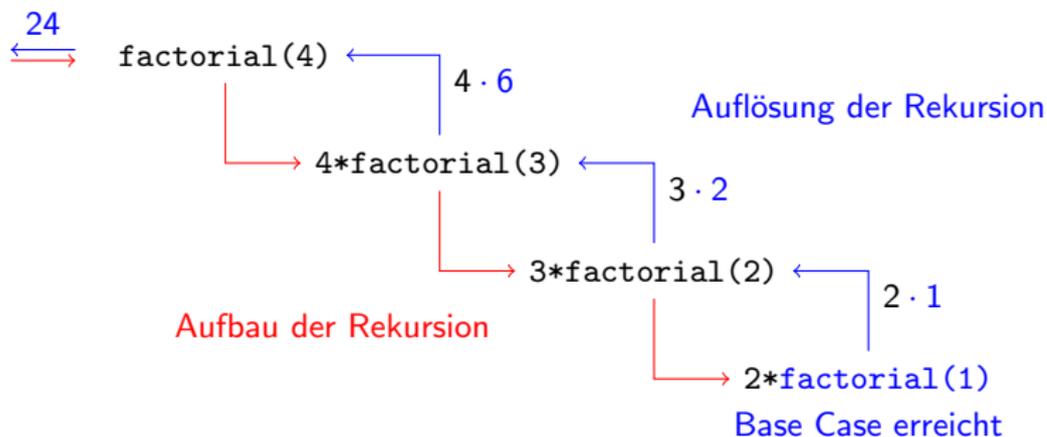
Da bei jedem Funktionsaufruf ein neuer Kontext für die Speicherung von Variablen angelegt werden muss, kostet die rekursive Lösung einer Aufgabe (im Gegensatz zur iterativen Lösung mit Schleifen) zusätzlichen Arbeitsspeicher. Deshalb wird Rekursion dann angewendet, wenn sich die Problemgröße bei jedem Funktionsaufruf um einen Faktor verkleinert (z. B. halbiert), so dass nur relativ wenige Aufrufe der Funktion nötig sind.

Der Vorteil der Rekursion gegenüber der Iteration besteht darin, dass sich bestimmte algorithmische Probleme damit einfacher lösen lassen – vorausgesetzt man versteht, wie Rekursion funktioniert.

Beispiel

```
1 def factorial(n): # Rekursive Berechnung von n! (Fakultät)
2     if n == 1:    # Base Case
3         return 1
4     else:         # Löse das nächstkleinere Problem ...
5         return n*factorial(n-1)
```

Rekursionsschema für factorial(4):



Aufgabe

Schreibe eine Funktion `factorial_iterative(n)`, die $n!$ für $n = 0, 1, 2, \dots$ iterativ, d. h. mit einer Schleife berechnet. Beachte, dass $0! = 1$ gilt.

Lösung der Aufgabe

```
1 def factorial(n): # Iterative Berechnung von n! (Fakultät)
2     f = 1
3     for k in range(2, n+1):
4         f = k * f
5     return f
```

Man muss hier einräumen, dass auch die iterative Lösung nicht besonders schwierig zu verstehen bzw. zu programmieren ist.