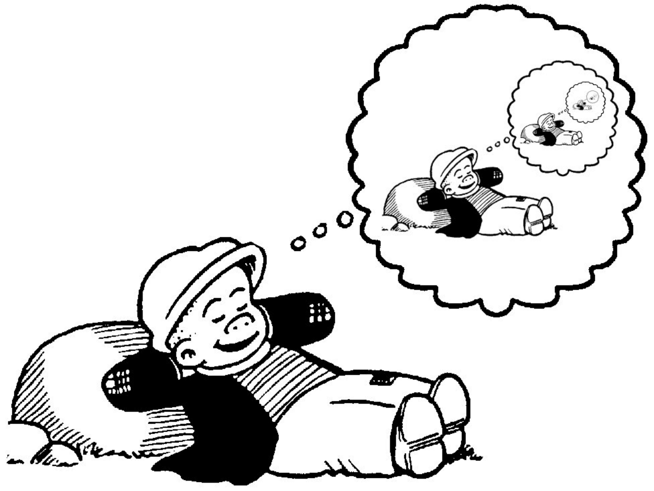
Rekursion

by

Dr. Günter Kolousek

Motivation



Quelle: `http://freakingid.com/`

Rekursive Akronyme

- Beispiele
  - GNU … GNU’s Not Unix
  - PHP … PHP Hypertext Processor
  - TikZ … TikZ Ist Kein Zeichenprogramm
- Konstruktionsmethode
  - Akronym ausdenken, z.B.: REE … Rekursion Einfach Erklärt
  - um einen Buchstaben erweitern: TREE
  - ganzes Akronym als ersten Teil verwenden: TREE … Tree Rekursion Einfach Erklärt

Begriff und Prinzip

- Definition Rekursion in der Mathematik und Informatik

  > *… in which the solution to each problem depends on the solutions to smaller instances of the same problem*
  > *– Graham, Ronald; Donald Knuth, Oren Patashnik (1990). Concrete Mathematics*

- Prinzip
  - Eine unendliche Menge von Objekten durch eine endliche Aussage definieren.
  - Eine unendliche Anzahl von Berechnungen durch einen endlichen Algorithmus (ohne Schleifen) zu beschreiben.

Beispiele

- Rekursive Definition von Mengen
  - Menge der ungeraden natürlichen Zahlen $\mathbb{N}_u$
    - $1 \in \mathbb{N}_u$
    - Wenn $x \in \mathbb{N}_u$, dann ist auch $(x+2) \in \mathbb{N}_u$
- Rekursive Alorithmen
  - Summe alle Zahlen von 1 bis n: $\text{sum}(n) = \sum_{i=1}^{n} i$
    - $\text{sum}(1) = 1$
    - $\text{sum}(n) = \text{sum}(n-1) + n$
- Rekursive Datenstrukturen
  - z.B. Bäume
    - O ist ein Baum (genannt leerer Baum)
    - Wenn t_1 und t_2 Bäume sind, dann ist auch die Struktur aus einem Knoten mit zwei Verzweigungen t_1 und t_2 ein Baum.

Funktionsweise und Umsetzung

- Es wird ein einfaches Problem spezifiziert und gelöst
- Es werden allgemeine Regeln angegeben, die alle anderen Probleme auf das einfache Problem zurückführen lässt.
- Umsetzung der Summenfunktion `sum(n)` in Python:

```python
def sum(n):
    if n == 1:      # Abbruchbedingung!
        return 1
    else:
        return sum(n - 1) + n
```

Ablauf für sum(4)

```
sum(4) = sum(3) + 4
    sum(3) = sum(2) + 3
        sum(2) = sum(1) + 2
            sum(1) = 1
        sum(2) = 1 + 2 = 3
    sum(3) = 3 + 3 = 6
sum(4) = 6 + 4 = 10
```

Faktorielle (Fakultät) $n!$

- Iterative Definition: Produkt aller nat. Zahlen von 1 bis n
  - $n! = 1 \cdot 2 \cdot \ldots \cdot n$
- Rekursive Definition: Produkt von $(n-1)!$ mit n, wobei 1! gleich 1 ist, d.h.:
  - $1! = 1$
  - $n! = (n-1)! \cdot n$
- Umsetzung in Python

```python
def fak(n):
    if n == 1:
        return 1
    else:
        return fak(n - 1) * n

return fak(1000)
```

Beispiel: Umdrehen eines Strings

```
def reversed(s):
    if s:
        return s[-1] + reversed(s[:-1])
    else:
        return ""
```

Und wie sieht die iterative Lösung aus?

Beispiel mit lokalen Variablen

```
def print_reversed():
    c = input("Bitte ein Zeichen ('.' beendet): ")
    if c != ".":
        print_reversed()
        print(c, end="")
```

```
>>> print_reversed()
Bitte ein Zeichen ('.' beendet): a
Bitte ein Zeichen ('.' beendet): b
Bitte ein Zeichen ('.' beendet): c
Bitte ein Zeichen ('.' beendet): .
cba
```

D.h. jeder Aufruf hat seine eigenen lokalen Variablen!

Rekursiver ggT

- Prinzip des euklidischen Algorithmus ($\leadsto$ Stäbe abschneiden):
  - Wenn $a = b$, dann ist das Ergebnis a
  - Wenn $a > b$, dann ist das Ergebnis für a und b dieselbe wie für $a - b$ und b.
- rekursive Umsetzung:

```python
def euklid1(a, b):
    if a == b:
        return a
    elif a > b:
        return euklid1(a - b, b)
    else:
        return euklid1(b - a, a)
```

Nachteil?

Rekursiver ggT – 2

- Verbesserung (die auch mit Stäben der Länge 0 zurecht kommt):

```
def euklid2(a, b):
    if a == 0:
        return b
    elif a > b:
        return euklid2(a - b, b)
    else:
        return euklid2(b - a, a)
```

Nachteile?

Vorteile und Nachteile

- Vorteile
  - einfache Formulierung von rekursiven Fragestellungen
    - dadurch auch bessere Performance möglich als bei eigener iterativer Umsetzung
- Nachteile
  - Performance $\rightsquigarrow$ Funktionsaufrufe
  - Ressourcenverbrauch $\rightsquigarrow$ Speicher für lokale Variablen, (auch Argumente), Rücksprungadresse
  - maximale Rekursionstiefe

Maximale Rekursionstiefe?

- `fak(1000)` →

```
Traceback (most recent call last):
File ”<stdin>”, line 11, in <module>
File ”<stdin>”, line 9, in main
File ”<stdin>”, line 7, in fak
...
File ”<stdin>”, line 7, in fak
File ”<stdin>”, line 7, in fak
File ”<stdin>”, line 4, in fak
RecursionError: maximum recursion depth \
  exceeded in comparison
```

- Stack!
  - lokale Variable
  - Rücksprungadresse

Iterativer ggT

```python
def slow_euklid(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```

Warum langsam? Verbesserung?

Iterativer ggT – 2

```
def fast_euklid1(a, b):
    if a < b:  # Stab a muss groesser sein!
        a, b = b, a
    while b > 0:
        r = a % b # wiederholtes Abziehen vermeiden
        a = b
        b = r
    return a
```

Kürzer möglich?

Iterativer ggT – 3

```python
def fast_euklid2(a, b):
    while b > 0:  # oder b != 0
        r = a % b
        a = b
        b = r
    return a
```

Dadurch ein Schleifendurchgang mehr…
Geht es noch kürzer?

Iterativer ggT – 4

```python
def fast_euklid3(a, b):
    while b:  # Abkuerzung fuer: b != 0
        # Elimination der Zwischenvariable:
        a, b = b, a % b
    return a
```

Kürzer geht es nicht mehr!

Rekursiver ggT – 3

Hmm,… kann man nicht die Tricks der iterativen Variante wiederverwenden?

```python
def euklid3(a, b):
    if a == 0:
        return b
    else:
        return euklid3(b, a % b)
```

Aber wie gesagt: in diesem Fall ist die iterative Variante viel vernünftiger!

Binäre Suche

- Problemstellung: Suche eines Datensatzes in einer sortierten direkt zugreifbaren Sequenz.
- Prinzip:
    1. vergleiche Suchschlüssel mit dem mittleren Eintrag
    2. wenn gleich, dann hat man den gesuchten Datensatz gefunden
    3. wenn kleiner, dann in der linken Hälfte weitersuchen
    4. anderenfalls in der rechten Hälfte weitersuchen

Binäre Suche – 2

```python
def binary_search(seq, key):
    left = 0
    right = len(seq) - 1
    while left <= right:
        mid = (left + right) // 2
        if seq[mid] == key:
            return mid
        elif seq[mid] > key:
            right = mid - 1
        else:
            left = mid + 1
    return None
```

- Wie sieht der rekursive Algorithmus aus?
- iterativ ist wiederum günstiger, aber wie sieht es aus mit…

Türme von Hanoi

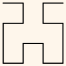
- Geschichte
  - 64 Scheiben, je Scheibe 1 Sekunde: 585 Milliarden Jahre!

```python
def hanoi(n, src, dst, aux):
    """Loese Tuerme von Hanoi mit n Tuermen
       von src nach dst ueber aux"""
    if n > 1:
        hanoi(n-1, src, aux, dst)
    print("Scheibe von", src, "nach", dst)
    if n > 1:
        hanoi(n-1, aux, dst, src)
```

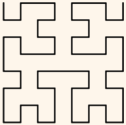
Hilbert-Kurve

- nach dem ”Erfinder” D. Hilbert (1891)
- Hilbertkurve H_1

- Hilbertkurve H_2

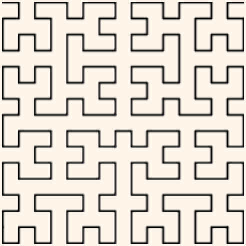


- Hilbertkurve H_3



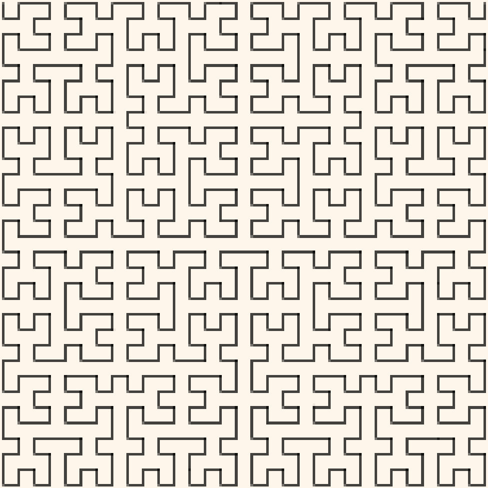
Hilbert-Kurve – 2

- Hilbertkurve H_4



Hilbert-Kurve – 3

- Hilbertkurve H_5



Hilbert-Kurve – 4

```
def hilbert(level, angle=90):
    if level:
        right(angle)
        hilbert(level - 1, -angle)
        forward(size)
        left(angle)
        hilbert(level - 1, angle)
        forward(size)
        hilbert(level - 1, angle)
        left(angle)
        forward(size)
        hilbert(level - 1, -angle)
        right(angle)
```

Reihenfolge von Befehlen

- vor dem Aufruf:

```
def print_str(s):
    print(s[0], end="")
    if s[1:]:
        print_str(s[1:])
```

liefert: abc

- nach dem Aufruf:

```
def print_str(s):
    if s[1:]:
        print_str(s[1:])
    print(s[0], end="")
```

liefert: cba

Theorie – Allgemeines

- Direkte Rekursion: Funktion f ruft sich selbst auf.
- Indirekte Rekursion: Funktion f ruft Funktion g auf, die wiederum Funktion f aufruft.
- Umsetzung von Rekursion in Iteration
  - Ist dies immer möglich?
    - Ja, aber nicht immer sinnvoll! Siehe Vorteile!
    - Jeder rekursiver Algorithmus in iterativer Variante möglich!

Endrekursion (tail recursion)

- Eine rekursive Funktion f ist *endrekursiv*, wenn der rekursive Funktionsaufruf die letzte Aktion zur Berechnung von f ist.
  - Die Funktion `euklid3` ist endrekursiv.
  - Die Funktion `fak` ist **nicht** endrekursiv.
- Eine endrekursive Funktion ist strikt *endrekursiv* (*strictly tail-recursive)*, wenn im Funktionsaufruf genau/nur die die formalen Parameter als Argumente übergeben werden.
  - Achtung: Oft wird der Begriff *endrekursiv* anstatt *strikt endrekursiv* verwendet.
- Strikt endrekursive Funktionen können in eine iterative Form gewandelt werden!
  - aber wie?

Umwandlung in iterative Form

```python
def fak(n):
    if n == 1:
        return 1
    else:
        return fak(n - 1) * n
```

nicht endrekursiv!

Umwandlung in iterative Form

```
def fak(n):
    if n == 1:
        return 1
    else:
        return fak(n - 1) * n
```

nicht endrekursiv!

da Multiplikation die letzte Operation darstellt

Umwandlung in iterative Form – 2

```python
def fak(n, acc=1):
    if n == 1:
        return acc
    else:
        return fak(n - 1, n * acc)
```

… Zwischenschritt: ein rekursiver Funktionsaufruf alleine (endrekursiv)

Umwandlung in iterative Form – 3

```python
def fak(n, acc=1):
    if n == 1:
        return acc
    else:
        acc = acc * n
        n = n - 1
        return fak(n, acc)
```

strikt endrekursiv!

Umwandlung in iterative Form – 4

```
def fak(n, acc=1):
    while (True):
        if n == 1:
            return acc
        else:
            acc = acc * n
            n = n - 1
```

iterativ!