Informatik-Propädeutikum

Dozentin: Dr. Claudia Ermel

Betreuer: Sepp Hartung, André Nichterlein, Clemens Hoffmann

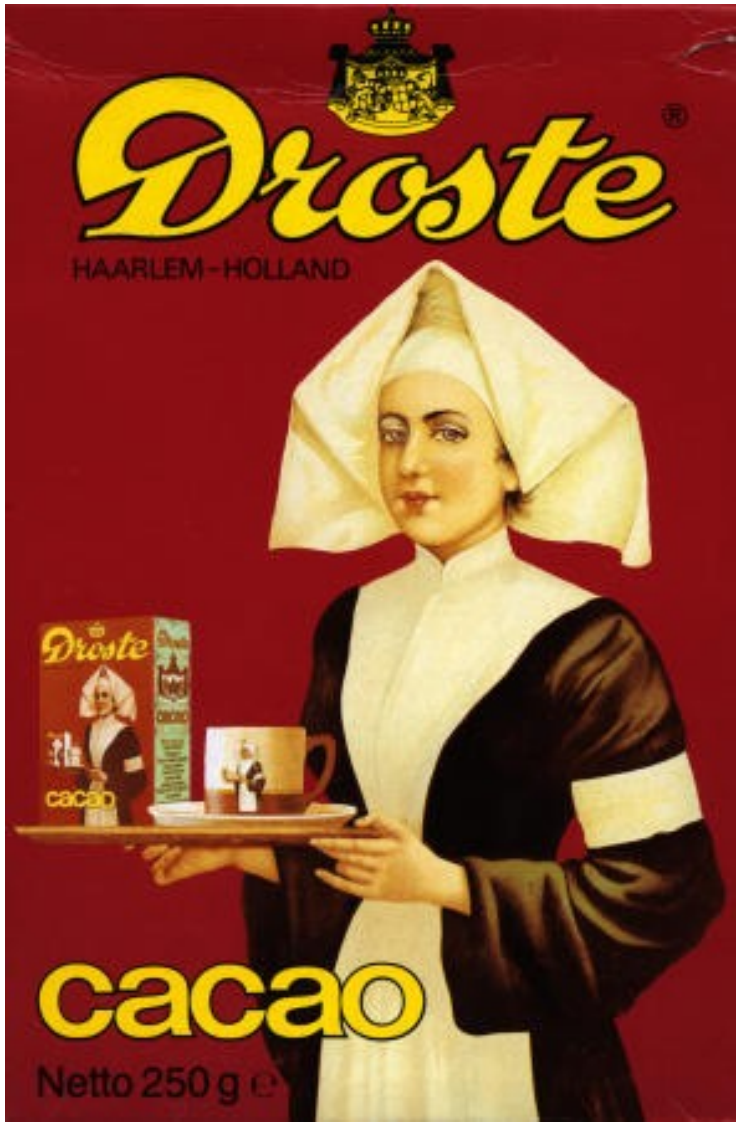
Sekretariat: Christlinde Thielcke (TEL 509b)

TU Berlin
Institut für Softwaretechnik und Theoretische Informatik
Prof. Niedermeier
Fachgruppe Algorithmik und Komplexitätstheorie
http://www.akt.tu-berlin.de

Wintersemester 2013/2014

Gliederung

Rekursion: Beispiele



Der „Droste-Effekt“, 1904

Eine rekursive Definition

Eine Matrjoschka besteht aus einer Puppe, die eine kleinere Matrjoschka enthält, oder es ist die kleinste Matrjoschka.

Was ist Rekursion?

Wikipedia: Als Rekursion (lateinisch recurrere „zurücklaufen“) bezeichnet man die Technik in Mathematik, Logik und Informatik, eine Funktion bzw. ein Objekt durch *einfachere Varianten von* sich selbst zu definieren.

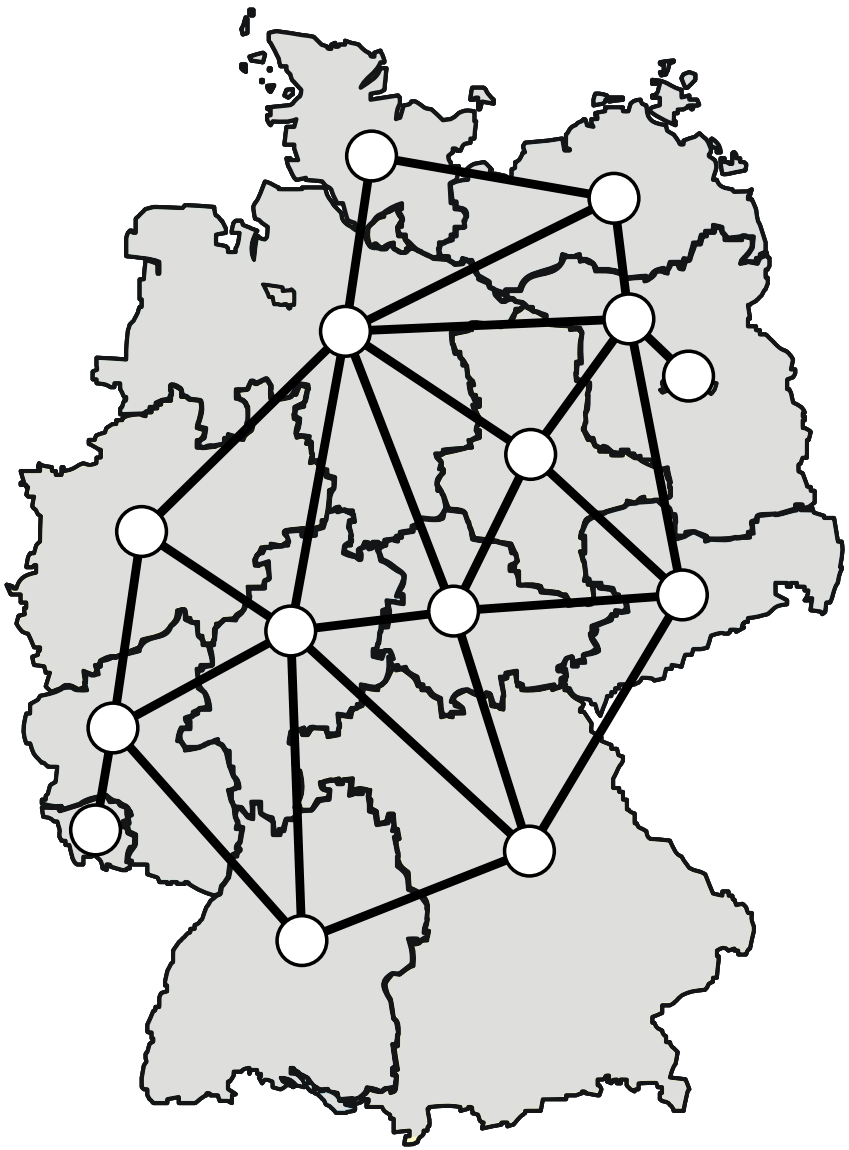
Beispiele:

1. Eine **Liste** besteht aus dem Erst-Element und der Rest-Liste, oder es ist die leere Liste.
2. Ein **Binärbaum** besteht aus einem Element (der Wurzel) und zwei binären Teil-Bäumen, oder es ist der leere Baum.

In der *Algorithmik* ist Rekursion eine mächtige „Reduktionstechnik“, der im wesentlichen folgende zwei Aspekte innewohnen:

- Falls eine Instanz eines Problems klein oder einfach genug ist, so löse sie einfach.
- Ansonsten führe das Problem auf eine oder mehrere *einfachere Instanzen des selben Problems* zurück.

Landkartenfärben revisited I



Beobachtung: Genaue Form der Bundesländer nicht wichtig, nur Nachbarschaftsrelation entscheidend.

Modellierung als Graphproblem:
Bundesländer $\hat{=}$ Knoten
Nachbarschaft $\hat{=}$ Kanten

Ziel: Finde Färbung der Knoten mit möglichst wenigen Farben, sodass keine zwei benachbarten Knoten die gleiche Farbe haben!

Vier-Farben-Satz: Jeder planare Graph ist vierfärbbar.

Landkartenfärben „revisited“ II

4-Färbung ist schwierig effizient zu finden, nicht aber 6-Färbung:

Konsequenz aus dem **Euler'schen Polyedersatz:** Für (einfache) planare Graphen gilt: Kantenzahl $\leq 3 \cdot$ (Knotenanzahl -2).

Folgerung: Jeder planare Graph hat einen Knoten mit max. fünf Nachbarn.

$\rightsquigarrow$**Rekursiver Algorithmus zum Berechnen einer 6-Färbung:**
Solange der Graph noch > 6 Knoten hat,

1. Finde einen Knoten v mit max. fünf Nachbarn,
2. Lösche v aus dem Graphen und färbe *rekursiv* den entstehenden Graphen mit höchstens sechs Farben.
3. Weise v eine der sechs Farben zu, die keiner von v's Nachbarn trägt.

Hinweise:

- Nach Löschen eines Knoten bleibt ein Graph planar.
- Da v nur fünf Nachbarn hat, so muss eine der sechs Farben reichen, um v verschieden von all seinen Nachbarn zu färben.

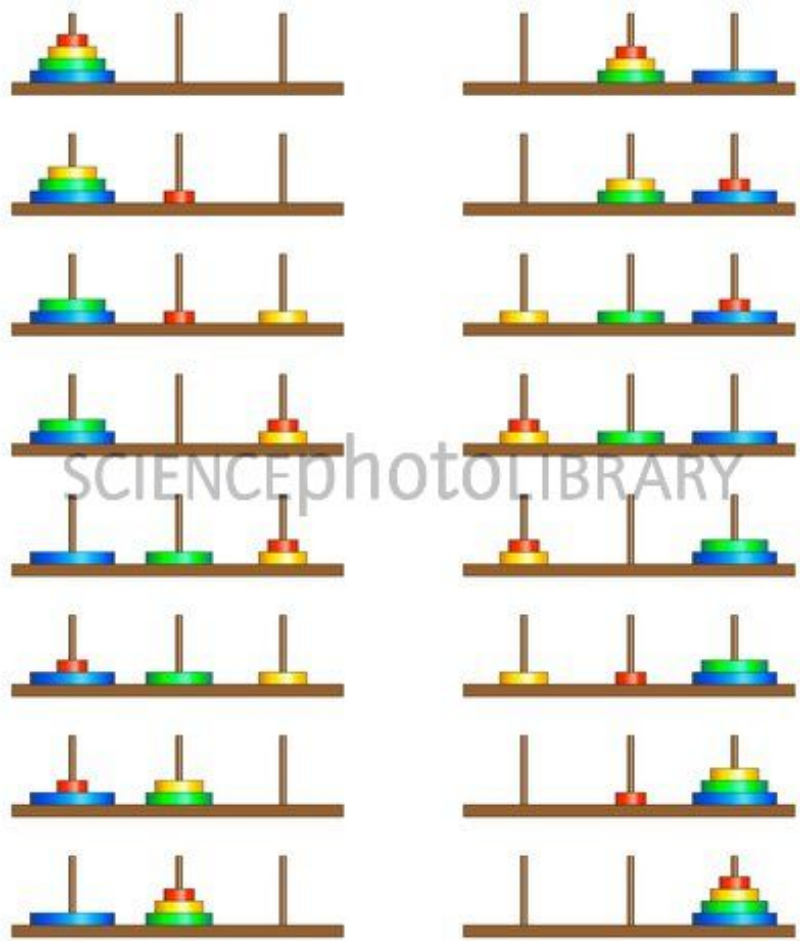
Türme von Hanoi I

Ein klassisches Beispiel für die Eleganz von Rekursion:

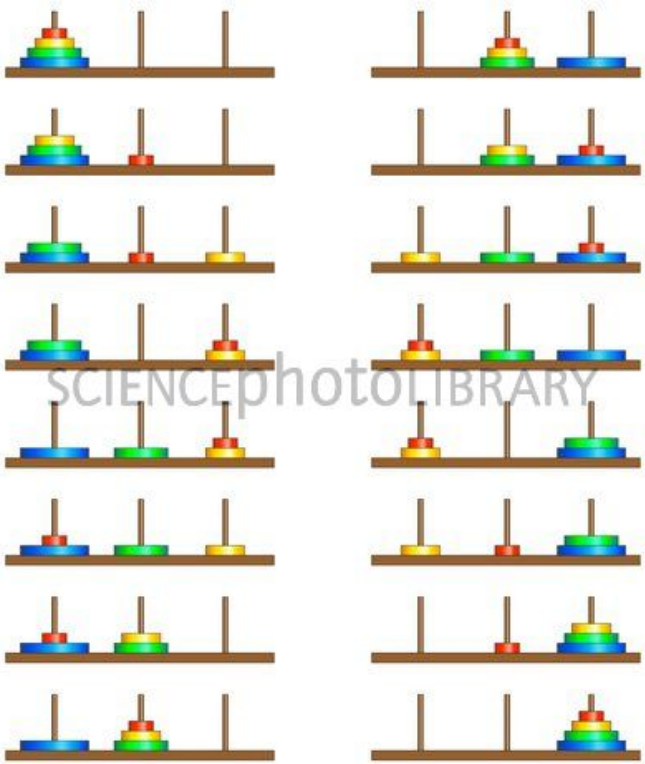
Gegeben n Scheiben (alle verschieden groß), die sich zu Beginn der Größe nach sortiert auf einem Stapel befinden.

Ziel ist nun die Bewegung des Stapels unter Zuhilfenahme eines „Zwischenstapels“ auf einen weiteren Stapel, wobei

- immer nur eine Scheibe transportiert werden kann und
- nie eine kleinere Scheibe unter einer größeren Scheibe liegen darf.



Türme von Hanoi II



Rekursive Lösungsidee: Zerteile das Bewegungsproblem in drei Phasen:

1. Bewege die oberen $n-1$ Scheiben zum Zwischenstapel ($\rightsquigarrow$ **Rekursion!**).
2. Bewege die letzte und größte Scheibe vom Start- zum Zielstapel.
3. Bewege alle Scheiben vom Zwischenstapel zum Zielstapel ($\rightsquigarrow$ **Rekursion!**).

Bemerkung: Wenn zur Bewegung von n Scheiben insgesamt $T(n)$ Einzelbewegungen nötig sind, so gilt $T(n) = 2 \cdot T(n-1) + 1 = 2^n - 1$, wobei $T(0) = 0$.

Bei $n = 64$ Scheiben würde man bei einer Scheibenbewegung pro Sekunde also 585 Milliarden Jahre brauchen...

Fibonacci-Zahlen I

Fibonacci-Zahlen II

Leonardo Fibonacci beschrieb 1202 das Wachstum einer Kaninchenpopulation mit Hilfe der Fibonacci-Folge.

Fibonacci-Folge: 0,1,1,2,3,5,8,13,21,34,...

Rekursives Konstruktionsprinzip: Die n-te Fibonacci-Zahl ergibt sich durch $F_n := \begin{cases} 0 & \text{falls } n = 0, \\ 1 & \text{falls } n = 1, \\ F_{n-1} + F_{n-2} & \text{falls } n > 1. \end{cases}$

$\rightsquigarrow$**Rekursiver Algorithmus zur Berechnung von F_n in Pseudocode**:

function $\mathrm{fib1}(n)$
1 **if** $n = 0$ **return** 0
2 **if** $n = 1$ **return** 1
3 **return** $\mathrm{fib1}(n-1) + \mathrm{fib1}(n-2)$

Fibonacci-Zahlen III

Wieviele Rechenschritte führt obiger Algorithmus aus?
Bei Eingabe n gibt es „offenbar“ F_n viele rekursive Aufrufe...
Können wir F_n *abschätzen*?

Ja! Anhand der Knotenanzahl im binären Baum der rekursiven Aufrufe!
Die Anzahl ist $2^{n+1} - 1$ (Beweis per vollst. Induktion, Tafel), also wächst die Zahl der rekursiven Aufrufe exponentiell in n und der rekursive Algorithmus ist somit hochgradig **ineffizient**!
Schnellster Supercomputer „Tian-he2“ (Juni 2013): 34 PetaFLOPS.
1 Rekursionsaufruf $\hat{=}$ 1 FLOP (Floating Point Operations Per Second)
$\rightsquigarrow$ Rechenzeit für F_{101}: < 0.5 Sekunden; F_{201}: > 1 Million Jahre!

Hoffnung (?): Bessere Hardware, sprich schnellere Rechner helfen!
„Moores Gesetz“: *Rechengeschwindigkeit verdoppelt sich alle 18 Monate.*
Leider hilft das nur unwesentlich bei exponentiell wachsenden Laufzeiten!

Aber es gibt eine viel wirksamere Lösung: **Iteration statt Rekursion** vermeidet überflüssige Mehrfachberechnung gleicher Zwischenergebnisse.

Fibonacci-Zahlen IV

Entrekursivierung (Iteration statt Rekursion) verhilft zu einer drastischen Effizienzsteigerung:

Zentrale Ineffizienz bei $\mathrm{fib1}(n)$:

Mehrfachberechnung ein und desselben Ergebnisses!

$\rightsquigarrow$ Idee: Speicherung von Zwischenergebnissen:

Iterativer Algorithmus zur Berechnung von F_n:

function $\mathrm{fib2}(n)$

```
1  if n = 0 return 0
2  create array f[0...n]
3  f[0] := 0; f[1] := 1
4  for i = 2, ..., n :
5      f[i] := f[i − 1] + f[i − 2]
6  return f[n]
```

Die Laufzeit von $\mathrm{fib2}(n)$ ist linear in n, d.h. es wird eine Anzahl von Rechenschritten benötigt, die direkt proportional zu n ist.

Tiefensuche I

Suche im Labyrinth:

- Aufgabe: Durchmustern eines Labyrinths.

- Präzisierung der Aufgabe:
  - Es soll sichergestellt werden, dass jede Kreuzung und jede Sackgasse irgendwann besucht wird.
  - Es soll verhindert werden, dass unbemerkt im Kreis gegangen wird.

Tiefensuche II (Durchmusterung eines Labyrinths)

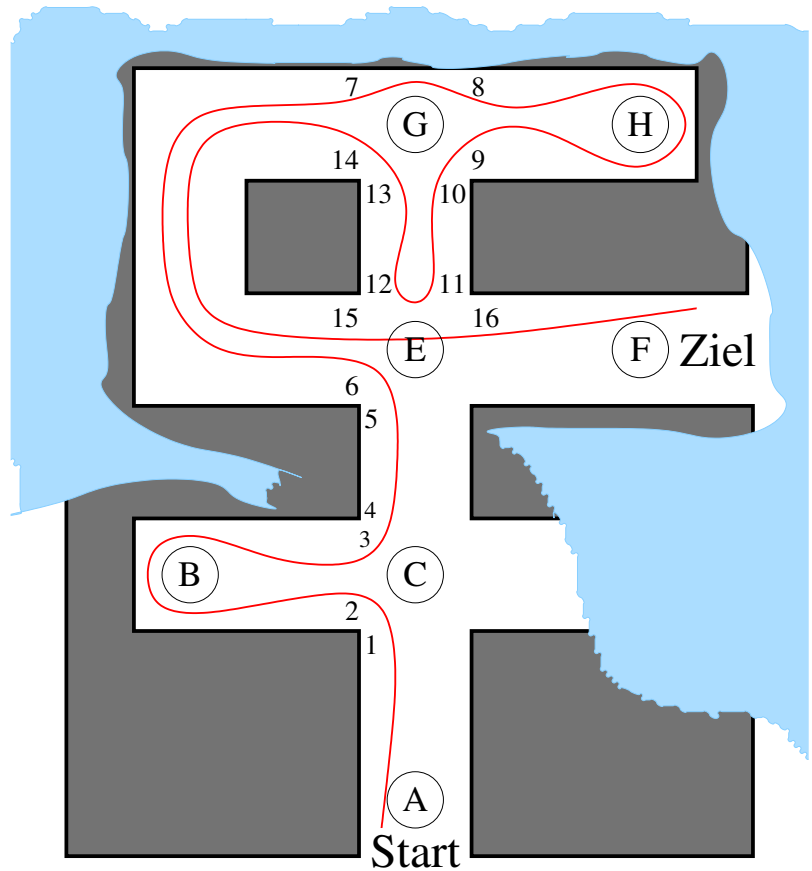
Lösung mit Hilfe von Markierungen an den Kreuzungen:

- Sackgasse: Umdrehen.
- Kreuzung: Beim Betreten: Markierung an die Wand des Ganges, über den man kam, malen. Außerdem:

  Nicht im Kreis laufen! Hat der Gang, durch den man gekommen ist, eben seine erste Markierung bekommen und sind weitere Markierungen vorhanden: Zweite Markierung malen und umdrehen.

  Sonst: Unerkundete Gänge suchen! Falls Gänge ohne Markierungen vorhanden: Davon den ersten von links nehmen, Markierung an die Wand malen.

  Sonst: Zurückgehen: Den Gang nehmen, der nur eine Markierung hat.



Tiefensuche III

- Grundidee:
  Immer möglichst tief in das Labyrinth hineingehen. Erst wenn dabei auf eine Sackgasse gestoßen wird, zur letzten Kreuzung zurückgehen und von dort aus erneut versuchen.

- Implementierung:

function *Tiefensuche*(X):
    if *Zustand*[X] = „entdeckt“ **then return**;
    if X = *Ziel* **then exit** „Ziel gefunden!“;
    Zustand[X] := „entdeckt“;
    for each benachbarte Kreuzung Y von X
        Tiefensuche(Y);

Tiefensuche IV

Eigenschaften

- Vorteile:
  - Einfachheit.
  - Effizienz.
- Nachteile:
  - Es wird nicht der kürzeste Weg zum Ziel gefunden.
  - In unendlich großen Labyrinthen wird evtl. ewig in der falschen Richtung gesucht.

Weitere Anwendungen

Tiefensuche kann nicht nur in Labyrinthen, sondern in beliebigen „labyrinth-artigen“ Suchräumen angewendet werden, z.B.

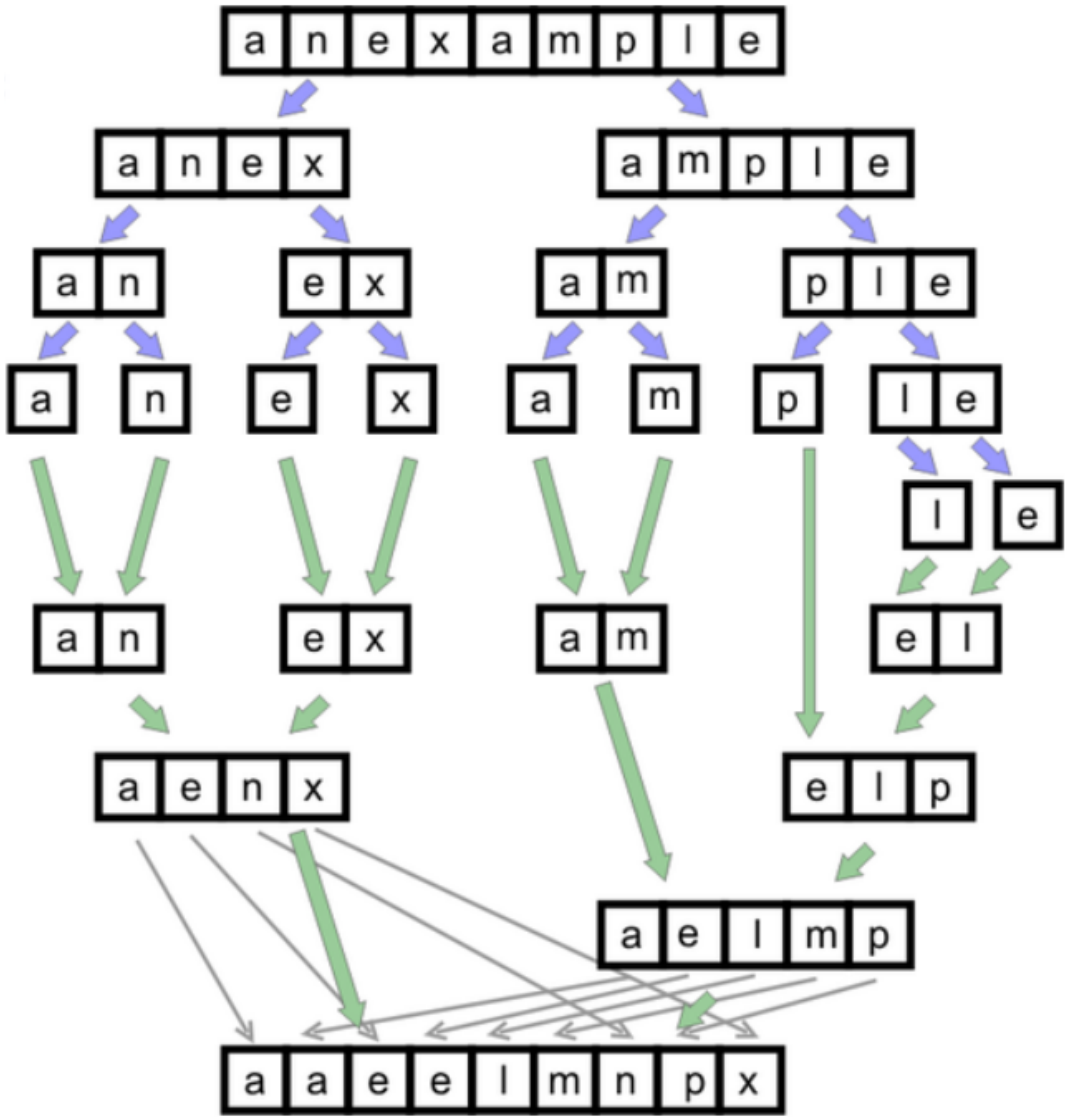
- in Labyrinthen mit „Einbahnstraßen“,
- im Internet,
- im „Labyrinth“ der Spielzüge bei Solitairespielen,
- ...

Aspekte der Rekursion in der Informatik

Folgende Aspekte von Rekursion sind in der Informatik besonders wichtig:

- Funktionale Programmierung (mit Vorteilen wie modularer Struktur und Freiheit von Nebeneffekten) beruht wesentlich auf Rekursion!
- Rekursive Datentypen (z.B. Listen bestehen aus Datum und Zeiger auf eine Liste).
- Um rekursive Algorithmen auf Rechnern zu realisieren benutzt das „Laufzeitsystem“ die Datenstruktur „Keller“.
- Oft erzielt man auch Effizienzgewinne durch „Entrekursivierung“ (vgl. Fibonacci-Zahlen).
- Wesentliche Rolle in der Theorie der Berechenbarkeit.
- Hilfsmittel zur Entwicklung eleganter und oft auch effizienter Algorithmen (wie z.B. Schnelle Fourier-Transformation durch Verwendung des Konzepts „Teile und Herrsche“).
- „To iterate is human, to recurse divine.“ – L. Peter **Deutsch**

Teile und Herrsche I:



Sortieren durch Verschmelzen (Merge Sort)

Teile und Herrsche II

Prinzip: Teile und Herrsche (Divide and Conquer) ist eine rekursive Programmiertechnik, bei der man das zu lösende Problem in kleinere Teilprobleme zerlegt, die Teilprobleme einzeln per Rekursion löst, und dann die erzielten „Teillösungen“ zu einer Gesamtlösung kombiniert.

Ein besonders **häufiger Spezialfall** (z.B. zutreffend für Sortieren durch Verschmelzen - “Mergesort”) ist, wenn man ein Problem in etwa *zwei gleich große Teilprobleme* zerlegt...; dies führt zu besonders effizienten Algorithmen. (Weitere Beispiele: Quicksort, Schnelle Fourier-Transformation.)

Bemerkung: Teile und Herrsche (lat. divide et impera) ist angeblich ein Ausspruch des französischen Königs Ludwigs XI. Er steht für das Prinzip, die eigenen Gegner und ihre Uneinigkeit für eigene Zwecke zu verwenden. Wichtig ist dabei, dass Uneinigkeit bei den Gegnern gefördert oder sogar verursacht wird, so dass diese in einzelnen, kleineren Gruppen leichter zu besiegen sind.

Teile und Herrsche III: Schnelles Potenzieren

Zur Berechnung der Zahl a^n für großes n verwende folgende Rekursion:

$$a^n := \begin{cases} 1 & \text{falls } n = 0, \\ (a^{n/2})^2 & \text{falls } n > 0 \text{ und } n \text{ gerade}, \\ a \cdot (a^{(n-1)/2})^2 & \text{falls } n > 0 \text{ und } n \text{ ungerade}. \end{cases}$$

Somit benötigt man nur $\log_2 n$ rekursive Aufrufe; dies ist für großes n viel effizienter als trivial $n - 1$ Multiplikationen auszuführen...

Teile und Herrsche IV: Schnelles Multiplizieren

Aufgabe: Multipliziere zwei n-Bit-Zahlen x und y:

Beispiel: $1100 \cdot 1101 = 10011100$.
Mit der naiven Schulmethode braucht man ca. n^2 Bitoperationen.

Rekursiver Ansatz führt für große n zu einem schnelleren Algorithmus:

Sei $x = x_1 \cdot 2^{n/2} + x_0$ (wobei x_1 die $n/2$ höherwertigen Bits und x_0 die $n/2$ niederwertigen Bits bezeichne) und analog $y = y_1 \cdot 2^{n/2} + y_0$.

Dann gilt:
$x \cdot y = (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) = x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 =$
$x_1 y_1 \cdot 2^n + ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$.

Zentrale Beobachtung: Der Effizienzgewinn beruht darauf, dass die Multiplikation zweier n-Bit-Zahlen i.w. auf die Multiplikation dreier (und nicht wie vielleicht naiv angenommen vierer) $n/2$-Bit-Zahlen zurückgeführt werden kann!
(Beachte hierbei, dass Multiplikationen viel „teurer“ sind als Additionen!)

Die Berechnungskraft von Rekursion I

Eine insbesondere in der Berechenbarkeitstheorie berühmte rekursive Funktion ist die **Ackermann-Funktion**. Version nach Schöning:

$$
\begin{aligned}
f(k,1) &= 2, \\
f(1,n) &= n+2 \text{ für } n>1, \\
f(k,n) &= f(k-1, f(k,n-1)) \text{ für } k,n>1.
\end{aligned}
$$

Wilhelm Ackermann, 1896-1962

Bemerkung: Ackermann war promovierter Mathematiklehrer; eine Erklärung, warum er als aktiver Wissenschaftler nicht die akademische Laufbahn einschlug, gibt folgendes Zitat seines Doktorvaters David Hilbert: „Oh, das ist wunderbar. Das sind gute Neuigkeiten für mich. Denn wenn dieser Mann so verrückt ist, daß er heiratet und sogar ein Kind hat, bin ich von jeder Verpflichtung befreit, etwas für ihn tun zu müssen.“

Die Berechnungskraft von Rekursion II

Die Ackermann-Funktion wächst extrem schnell und ist ein berühmtes Beispiel für eine nicht „primitiv-rekursive“ Funktion.

$$\begin{aligned} f(k,1) &= 2, \\ f(1,n) &= n+2 \text{ für } n>1, \\ f(k,n) &= f(k-1,f(k,n-1)) \text{ für } k,n>1. \end{aligned}$$

Es gilt: $f(2,n)=2n; \quad f(3,n)=2^n; \quad f(4,n)=\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{n\text{-mal}}$

Beispiele:

$f(5,1)=2.$

$f(5,2)=f(4,f(5,1))=f(4,2)=f(3,2)=f(2,2)=f(1,2)=4.$

$f(5,3)=f(4,f(5,2)))=f(4,4)=\ldots=2^{2^{2^2}}=65536.$

$f(5,4)=f(4,f(5,3))=f(4,65536)=\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{65536\text{-mal}}.$

Die Berechnungskraft von Rekursion III

Nachfolgend betrachten wir Funktionen über den natürlichen Zahlen. Wir erklären zunächst einige einfache Funktionen quasi „per Festlegung“ („per Axiom“) als „berechenbar“: Dazu gehören insbesondere die *konstantwertigen* Funktionen und die „*Nachfolgerfunktion*“ $f(x) = f(x) + 1$. Weiterhin dürfen Funktionen ineinander *eingesetzt* werden. D.h. sind f und g berechenbar, so auch $f(g(x))$.

Primitive Rekursion entspricht der **Iteration**:
Wenn eine Funktion f bereits „berechenbar“ ist, so soll es auch die zweistellige Funktion $g(n, x) := f(f(\ldots f(x)\ldots))$ sein, wobei f n-mal hintereinander ausgeführt wird.

Iterativ ließe sich das so mit einer FOR-Schleife ausdrücken:
INPUT n, x; $y := x$; FOR $i := 1$ TO n DO $y := f(y)$;
OUTPUT y.

Die Berechnungskraft von Rekursion IV

Ein mächtigeres Werkzeug als die primitive Rekursion ist die μ-**Rekursion**, bei der im Wesentlichen FOR-Schleifen durch die mächtigeren WHILE-Schleifen ersetzt werden:

Wenn $f(n, x)$ berechenbar ist, dann auch $g(x) := \min\{n \mid f(n, x) = 0\}$.

Mit einer WHILE-Schleife lässt sich das wie folgt ausdrücken:
INPUT x; $n := 0$; WHILE $f(n, x) \neq 0$ DO $n := n + 1$;
OUTPUT n.

Mitteilung: Mit μ-Rekursion lässt sich die Ackermann-Funktion berechnen, nicht aber mit primitiver Rekursion. Weiterhin besteht die Hypothese, dass alles was berechenbar ist, auch mit μ-Rekursion berechenbar ist.