

## 4 Rekursion

### 4.1 Ein allgemeines Problemlösungsschema

### 4.2 Gray-Codes

### 4.3 Rekursive Funktionen

### 4.4 Partitionen

### 4.5 Türme von Hanoi

### 4.6 Multiplikation langer Zahlen

### 4.7 Binäre Suche

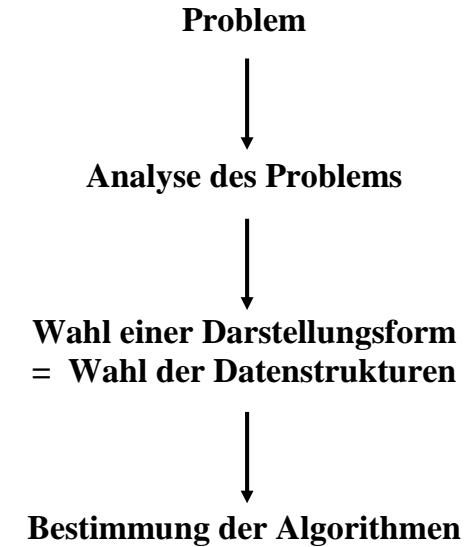
### 4.8 Permutationen

### 4.9 Quicksort

### 4.10 Lineare verkettete Liste

### 4.11 Lineare Rekursion

## Allgemeines Problem-Lösungs-Schema



### Bemerkungen:

- (i) Die Datenstrukturen verkörpern die Theorie zur Lösung eines Problems.
- (ii) Die Algorithmen sind in dieser Betrachtungsweise nur zweitrangig, sie steuern die Details zur Lösung eines Problems bei.
- (iii) Die Datenstrukturen bestimmen die Effizienz und Effektivität der benutzten Algorithmen.

Frage: Existieren nicht auch einfachere Wege?

## Rekursion als allgemeines Problemlösungsschema:

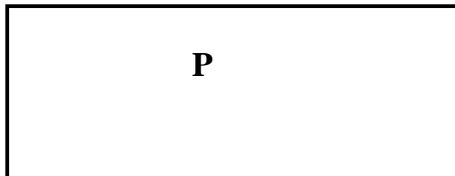
Ein erfolgreiches Grundmuster zur Lösung von Problemen läßt sich etwa so beschreiben:

1. Zerlege Problem P in kleinere Teilprobleme T1, T2, ... Tn.
2. Löse Teilprobleme T1, T2, ... Tn.
3. Konstruiere Lösung von P aus den Lösungen von T1, T2, ... Tn.

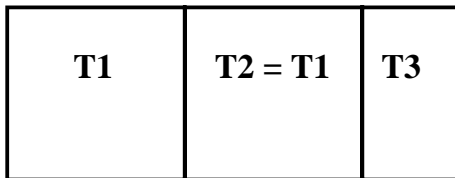
**Bemerkung:** Obiges Verfahren ist besonders ökonomisch, falls die Probleme P, T1, ... Tn von ähnlicher Struktur sind; dann läßt sich der Zerlegungsschritt wiederholen.

**Bildliche Illustration:**

**Problem:**



**Teilprobleme:**



## Gray-Codes:

**Beispiel eines 4-Bit Gray-Codes:**

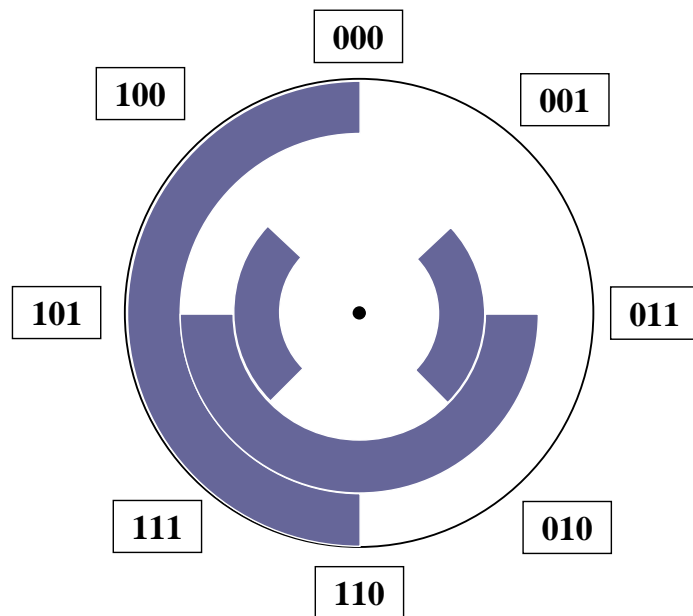
0 0 0 0	=	0
0 0 0 1	=	1
0 0 1 1	=	2
0 0 1 0	=	3
0 1 1 0	=	4
0 1 1 1	=	5
0 1 0 1	=	6
0 1 0 0	=	7
1 1 0 0	=	8
1 1 0 1	=	9
1 1 1 1	=	10
1 1 1 0	=	11
1 0 1 0	=	12
1 0 1 1	=	13
1 0 0 1	=	14
1 0 0 0	=	15

**Definition:** Eine Folge aller  $2^n$  Bitketten der Länge n ( $n \geq 1$ ) heißt ein Gray-Code der Länge n, falls sich jeweils 2 benachbarte Bitketten nur in einer Bitposition unterscheiden.

### Nutzung eines Gray-Codes bei der Winkelbestimmung:

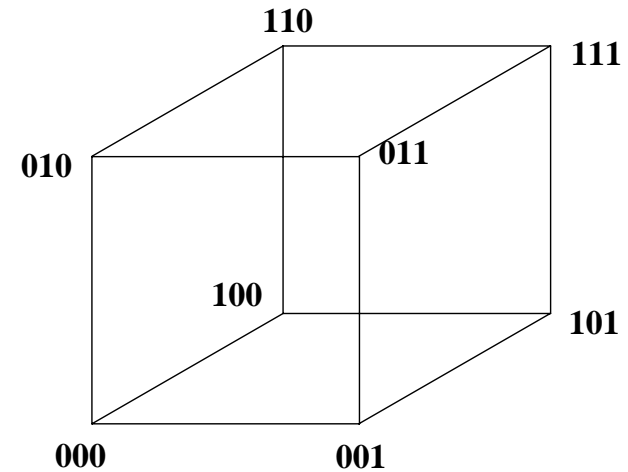
Der Namensgeber des Gray-Codes ist der Ingenieur Frank Gray, der für die Nutzung des Gray-Codes im Jahre 1953 das U. S. Patent 2 632 058 "Pulse Code Communication" erhielt. Gray-Codes wurde schon früher genutzt, z. B. im Jahre 1878 von Émile Baudot in der Telegraphie. Auch heute wird der Gray-Code verwendet, um z. B. Windrichtungen und Windgeschwindigkeiten zu bestimmen.

### Beispiel für 3-Bit-Code:

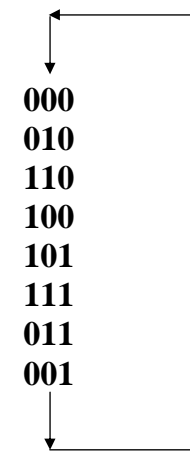


### Gray als Hamiltonkreis auf einem mehrdimensionalen Würfel:

### Beispiel: 3-dimensionaler Würfel



### Beispiel für Gray-Code:



### Beobachtung:

Ist  $x$  ein Gray-Code der Länge  $n$ , dann ist  $x$  reversiert auch ein Gray-Code der Länge  $n$ . Dies führt sofort zu einem Bildungsgesetz für eine Klasse von Gray-Codes.

#### Gray-Code der Länge 1:

0  
1

#### Gray-Code der Länge 2:

00  
01  
11  
10

#### Gray-Code der Länge 3:

000  
001  
011  
010  
110  
111  
101  
100

**Bemerkung:** Das hinzugefügte Bit beim Übergang vom Code der Länge  $n$  zu einem Code der Länge  $n+1$  ist kursiv dargestellt.

```
// Formulierung obigen Algorithmus zur Erzeugung  
// eines Gray-Codes als C++ - Routine
```

```
const int codelaenge = ...; // Länge des Gray-Codes  
  
int gray [codelaenge] = {0}; // enthält aktuelles  
// Codewort
```

```
// Routine zur Erzeugung des reflektierten Gray-Codes
```

```
void buildgray (int i) {  
    if (i >= 0) {  
        buildgray (i-1);  
        gray [i] ^= 1;  
        // Hier steht das aktuelle Codewort  
        // zur Bearbeitung zur Verfügung.  
        printgray ();  
        buildgray (i-1);  
    }  
}  
} //buildgray
```

```
// Beispielroutine zur Bearbeitung des  
// aktuellen Codeworts
```

```
void printgray () {  
    static int zahlwert = 0;  
    cout << setw (6) << zahlwert << ": ";  
    for (int i = codelaenge-1; i >= 0; --i)  
        cout << gray [i] << " ";  
    cout << endl;  
    ++zahlwert;  
} //printgray
```

## Reflektierter Gray-Code der Ordnung 5:

0:	0	0	0	0	0
1:	0	0	0	0	1
2:	0	0	0	1	1
3:	0	0	0	1	0
4:	0	0	1	1	0
5:	0	0	1	1	1
6:	0	0	1	0	1
7:	0	0	1	0	0
8:	0	1	1	0	0
9:	0	1	1	0	1
10:	0	1	1	1	1
11:	0	1	1	1	0
12:	0	1	0	1	0
13:	0	1	0	1	1
14:	0	1	0	0	1
15:	0	1	0	0	0
16:	1	1	0	0	0
17:	1	1	0	0	1
18:	1	1	0	1	1
19:	1	1	0	1	0
20:	1	1	1	1	0
21:	1	1	1	1	1
22:	1	1	1	0	1
23:	1	1	1	0	0
24:	1	0	1	0	0
25:	1	0	1	0	1
26:	1	0	1	1	1
27:	1	0	1	1	0
28:	1	0	0	1	0
29:	1	0	0	1	1
30:	1	0	0	0	1
31:	1	0	0	0	0

## Beispiele rekursiver Definitionen:

Fakultät:  $\forall n \in \mathbf{N}$ :

$$n! = \begin{cases} n \cdot (n-1)! & (n > 0) \\ 1 & (n = 0) \end{cases}$$

Potenz:  $\forall y \in \mathbf{N}, \forall x \in \mathbf{R}$ :

$$x^y = \begin{cases} x \cdot x^{y-1} & (y > 0) \\ 1 & (y = 0) \end{cases}$$

Fibonacci-Zahlen:  $\forall n \in \mathbf{N}$ :

$$f_n = \begin{cases} 1 & (n = 0 \vee n = 1) \\ f_{n-1} + f_{n-2} & (n > 1) \end{cases}$$

Geradzahligkeit:  $\forall n \in \mathbf{N}$ :

$$\text{even}(n) = \begin{cases} \text{true} & (n = 0) \\ \text{odd}(n-1) & (n > 0) \end{cases}$$

$$\text{odd}(n) = \begin{cases} \text{false} & (n = 0) \\ \text{even}(n-1) & (n > 0) \end{cases}$$

## Beispiele einfacher Funktionen:

### Definition:

```
function fac (n) ≡ if n = 0
                    then 1
                    else n * fac (n-1)
                    fi
```

### Aufruf:

```
fac (3) ≡ if 3 = 0
           then 1
           else 3 * fac (2)
           fi
≡ 3 * if 2 = 0
       then 1
       else 2 * fac (1)
       fi
≡ 3 * 2 * if 1 = 0
           then 1
           else 1 * fac (0)
           fi
≡ 3 * 2 * 1 * if 0 = 0
               then 1
               else 0 * fac (-1)
               fi
≡ 3 * 2 * 1 * 1
≡ 6
```

**Bemerkung:** Der Ausdruck `fac (-1)` muß nicht definiert sein, da er nicht ausgewertet wird.

### Definition:

```
function amult (a, b)
≡ if b = 0
  then 0
  else if even (b)
        then 2 * amult (a, b div 2)
        else a + 2 * amult (a, b div 2)
  fi
fi
```

### Aufruf:

```
amult (24, 10)
≡ 2 * amult (24, 5)
≡ 2 * (24 + 2 * amult (24, 2))
≡ 2 * (24 + 2 * (2 * amult (24, 1)))
≡ 2 * (24 + 2 * (2 * (24 + amult (24, 0))))
≡ 2 * (24 + 2 * (2 * (24 + 0)))
≡ 240
```

### Beispiel für eine Schar rekursiver Funktionen:

### Definition:

```
function ggt (x, y) ≡ if y = 0
                      then x
                      else ggt (y, mod (x, y))
                      fi
```

```
function mod (x, y) ≡ if x < y
                      then x
                      else mod (x-y, y)
                      fi
```

```
//
// Die Ackermann Funktion als Beispiel einer nicht-
// primitiven rekursiven Funktion als C++ - Routine:
//
// Für m, n aus N sei:
//   f(0, n) = n + 1
//   f(m+1, 0) = f(m, 1)
//   f(m+1, n+1) = f(m, f(m+1, n))
//
```

```
unsigned long ackermann ( unsigned long x,
                          unsigned long y) {
    if (x == 0)
        return y + 1;
    else if (y == 0)
        return ackermann (x-1, 1);
    else
        return ackermann (x-1, ackermann (x, y-1));
} //ackermann
```

**Bemerkung:** Die Ackermann-Funktion eignet sich hervorragend, um die Effektivität der Implementation des Prozeduraufrufs in Programmiersprachen zu testen.

### Einige Werte der Ackermannfunktion:

ackermann (0, 0) = 1	Aufrufe = 1
ackermann (0, 1) = 2	Aufrufe = 1
ackermann (0, 2) = 3	Aufrufe = 1
ackermann (1, 0) = 2	Aufrufe = 2
ackermann (1, 1) = 3	Aufrufe = 4
ackermann (1, 2) = 4	Aufrufe = 6
ackermann (1, 3) = 5	Aufrufe = 8
ackermann (2, 0) = 3	Aufrufe = 5
ackermann (2, 1) = 5	Aufrufe = 14
ackermann (2, 2) = 7	Aufrufe = 27
ackermann (2, 3) = 9	Aufrufe = 44
ackermann (2, 4) = 11	Aufrufe = 65
ackermann (2, 5) = 13	Aufrufe = 90
ackermann (2, 6) = 15	Aufrufe = 119
ackermann (2, 7) = 17	Aufrufe = 152
ackermann (2, 8) = 19	Aufrufe = 189
ackermann (2, 9) = 21	Aufrufe = 230
ackermann (2, 10) = 23	Aufrufe = 275
ackermann (2, 11) = 25	Aufrufe = 324
ackermann (3, 0) = 5	Aufrufe = 15
ackermann (3, 1) = 13	Aufrufe = 106
ackermann (3, 2) = 29	Aufrufe = 541
ackermann (3, 3) = 61	Aufrufe = 2.432
ackermann (3, 4) = 125	Aufrufe = 10.307
ackermann (3, 5) = 253	Aufrufe = 42.438
ackermann (3, 6) = 509	Aufrufe = 172.233
ackermann (3, 7) = 1021	Aufrufe = 693.964
ackermann (3, 8) = 2045	Aufrufe = 2.785.999
ackermann (3, 9) = 4093	Aufrufe = 11.164.370
ackermann (3, 10) = 8189	Aufrufe = 44.698.325
ackermann (3, 11) = 16381	Aufrufe = 178.875.096

## Partitionen:

**Frage:** Auf wie viele Arten lässt sich eine natürliche Zahl  $x$  als Summe vorgegebener Summanden darstellen?

**Beispiel:** Summanden: 1, 4, 25  
Zahl: 29

$$\begin{aligned} 29 &= 25 + 4 \\ &= 25 + 4 * 1 \\ &= 7 * 4 + 1 \\ &= 6 * 4 + 5 * 1 \\ &= 5 * 4 + 9 * 1 \\ &= 4 * 4 + 13 * 1 \\ &= 3 * 4 + 17 * 1 \\ &= 2 * 4 + 21 * 1 \\ &= 4 + 25 * 1 \\ &= 29 * 1 \end{aligned}$$

**Antwort:** 10 verschiedene Arten

**Bemerkung:** Man sieht sofort die rekursive Struktur der Antwortgenerierung.

## Partitionierung einer natürlichen Zahl:

Sei  $n \in \mathbb{N}$  mit  $n \geq 1$ , bestimme alle Zerlegungen  $x_1, x_2, \dots, x_i$  von  $n$  mit  $x_k \geq 1$  ( $k = 1 \dots i$ ) und

$$\sum_{k=1}^i x_k = n.$$

**Beispiel:**  $n = 5$

$$\begin{array}{cccccc} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & \\ 1 & 1 & 3 & & \\ 1 & 2 & 2 & & \\ 1 & 4 & & & \\ 2 & 3 & & & \\ 5 & & & & \end{array}$$

## Rekursionsschema zur Gewinnung aller Partitionen:

Gesamtheit aller Partitionen von  $n$

= alle Partitionen mit kleinstem Element 1

∪ alle Partitionen mit kleinstem Element 2

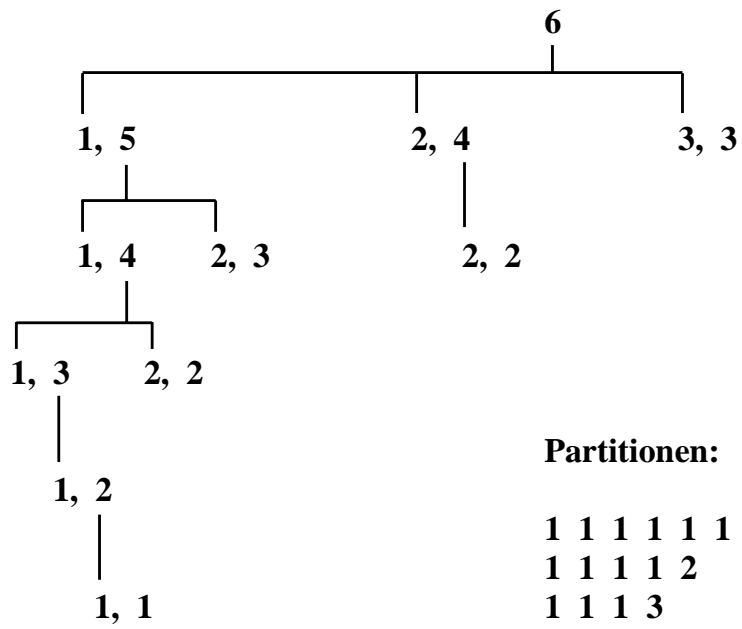
...

∪ alle Partitionen mit kleinstem Element  $\lfloor n/2 \rfloor$

∪ einelementiger Partition  $n$

**Bemerkung:** Nach Abspaltung von 1, 2, 3, ... hat man die Aufgabe, die Partition für eine kleinere Zahl zu bestimmen.

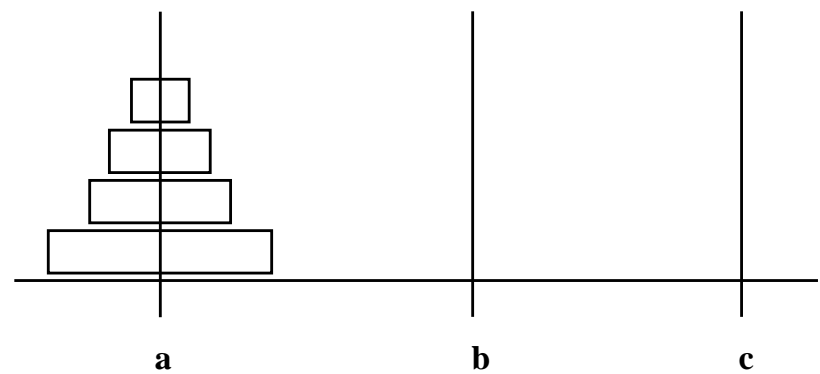
**Beispiel eines Partitionsbaums:**



**Partitionen:**

- 1 1 1 1 1 1
- 1 1 1 1 2
- 1 1 1 3
- 1 1 2 2
- 1 1 4
- 1 2 3
- 1 5
- 2 2 2
- 2 4
- 3 3
- 6

**Beispiel: Türme von Hanoi**



**Aufgabe:**

Man habe einen Stapel von  $n$  ( $n \geq 1$ ) aufeinanderliegenden unterschiedlich großen Scheiben am Ort a; der Stapel ist zum Ort b zu bewegen unter folgenden Randbedingungen:

- (i) Es darf zu einem Zeitpunkt nur eine freie Scheibe bewegt werden; eine Scheibe ist frei, falls sie auf einem Stapel zuoberst liegt.
- (ii) Es darf nur ein Hilfsstapel (am Ort c) gebildet werden.
- (iii) Niemals darf eine größere Scheibe auf einer kleineren liegen.

Für dieses Problem läßt sich sofort eine rekursive Lösung angeben.

```

proc Bewege Turm (integer Anzahl, // Turmhöhe
/* von */ Ort a, // Ausgangsort
/* nach */ Ort b, // Zielort
Ort c) // Zwischenort

if Anzahl = 1 then
    Bewege Scheibe von a nach b
else
    Bewege Turm (Anzahl - 1, a, c, b)
    Bewege Scheibe von a nach b
    Bewege Turm (Anzahl - 1, c, b, a)
fi
procend

```

**Bemerkung:** Die Zahl der Scheibenbewegungen ist  $2^n - 1$ , was man leicht rekursiv beweist.

## Multiplikation langer nichtnegativer Ganzzahlen:

**Annahme: Binärdarstellung der Zahlen**

**n-bittige Zahl Z, n gerade**

$$Z = \underbrace{\boxed{A}}_{n/2} \underbrace{\boxed{B}}_{n/2}$$

$$Z = A * 2^{n/2} + B$$

$$Y = C * 2^{n/2} + D \quad \text{sei eine weitere n-bittige Zahl.}$$

**Damit:**

$$Z * Y = A * C * 2^n + (A * D + B * C) * 2^{n/2} + B * D$$

**Aufwandsbetrachtung, falls nur die Multiplikation als aufwendige Operation angesehen wird.**

Sei nun n eine Potenz von 2, dann

$$L(1) = 1$$

$$L(n) = 4 * L(n/2) + \text{Konstante} * n$$

**Für eine grobe Aufwandsabschätzung genügt die Betrachtung der vereinfachten Rekursion:**

$$L(1) = 1$$

$$L(n) = 4 * L(n/2)$$

Da  $n = 2^x$  gefordert wurde, erhält man als Lösung:

$$L(n) = 4^x = 2^x * 2^x = n^2$$

**Bemerkung:** Die Komplexitätsklasse für die rekursive Multiplikation ist die gleiche wie für die gewöhnliche Multiplikation.

Eine Umformulierung des Produktes  $Z * Y$  liefert:

$$\begin{aligned} Z * Y &= A * C * 2^n \\ &+ ((A - B) * (D - C) + A * C + B * D) * 2^{n/2} \\ &+ B * D \end{aligned}$$

Die Aufwandsabschätzung unter der Annahme, daß nur die Multiplikation eine aufwendige Operation ist, führt zu:

$$\begin{aligned} L(1) &= 1 \\ L(n) &= 3 * L(n/2) + \text{Konstante} * n \end{aligned}$$

Die Lösung der vereinfachten Rekursion liefert nun:

$$L(n) = 3^x = (2^{\log_2 3})^x = (2^x)^{\log_2 3} = n^{\log_2 3} = n^{1,585}$$

Auch die Lösung der vollständigen Rekursion ist von der Ordnung  $n^{\log_2 3}$ .

Zum Vergleich:

$$\begin{aligned} 1000^2 &= 1.000.000 \\ 1000^{\log_2 3} &= 56.870,6 \end{aligned}$$

## Beispiel: Binäre Suche

Die Elemente  $x_1$  bis  $x_n$  mögen sortiert in einem Array  $a[1..n]$  liegen. Es ist zu prüfen, ob sich ein Element  $x$  im Array  $a$  befindet, falls ja, ist seine Positionsnummer zurückzugeben, falls nein, ist 0 zurückzugeben.

Eine mögliche Lösung läßt sich als rekursive Funktion formulieren:

```
function suche (element x,
                integer ug, // untere Suchgrenze
                integer og) // obere Suchgrenze
≡ (integer m := (ug + og) div 2;
   if ug > og then
       return 0
   elsif x = a[m] then
       return m
   elsif x < a[m] then
       return suche(x, ug, m-1)
   else
       return suche(x, m+1, og)
fi)
```

**Bemerkung:** Die rekursive Suche ist von logarithmischem Aufwand in der Zahl der Array-Elemente.

**Permutationen:**

**Annahme:** Die zu permutierenden Elemente mögen sich im Array  $a[1 \dots n]$  befinden.

**Rekursive Zerlegung:**

Erzeuge alle Permutationen von  $a[1]$  bis  $a[n-1]$ ,  
wobei  $a[n]$  fest bleibt.

Für  $k$  von 1 bis  $n-1$

Wiederhole

"Stelle Ausgangskonstellation wieder her"

tausche  $a[k]$  mit  $a[n]$

erzeuge alle Permutationen von  $a[1]$  bis  $a[n-1]$

Ende

**Beispiel: Alle Permutationen von 4 Elementen:**

A B C D      Ausgangspermutation, D fest

B A C D

C B A D

B C A D

A C B D

C A B D

A B C D      Herstellen der Ausgangs-  
permutation, nicht zählen

D B C A  
B D C A  
C B D A  
B C D A  
D C B A  
C D B A

A wurde mit D getauscht,  
Block von 6 Permutationen

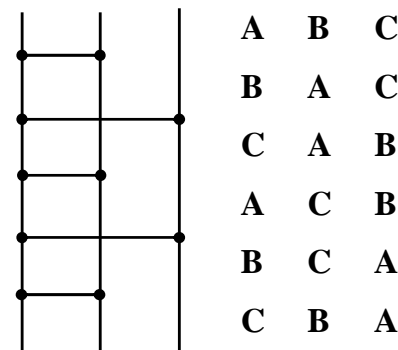
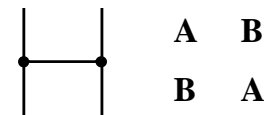
A D C B  
...  
B A D B

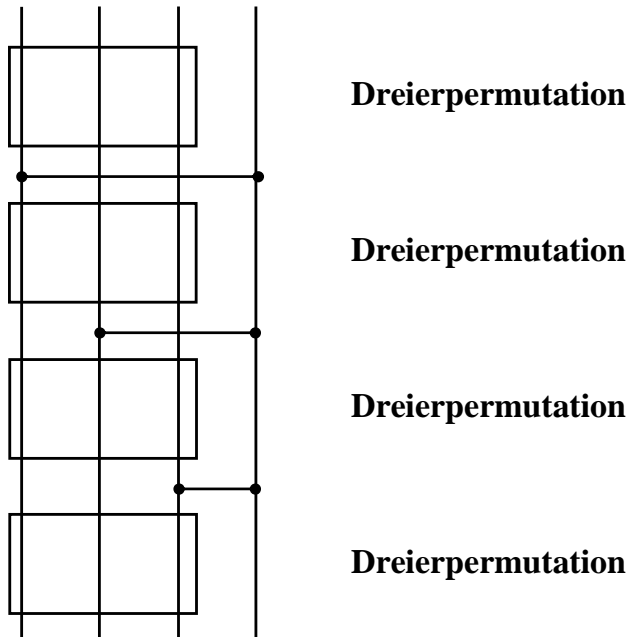
Block von 6 Permutationen

A B D C  
...  
D A B C

letzter Block von 6 Permutationen

**Tauschmuster nach Heap:**





**Erzeugung der 24 Permutationen von 4 Elementen nach Heap**

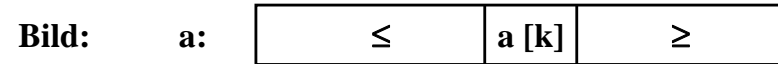
- |         |         |
|---------|---------|
| A B C D | A C D B |
| B A C D | C A D B |
| C A B D | D A C B |
| A C B D | A D C B |
| B C A D | C D A B |
| C B A D | D C A B |
| D B A C | D C B A |
| B D A C | C D B A |
| A D B C | B D C A |
| D A B C | D B C A |
| B A D C | C B D A |
| A B D C | B C D A |

**Beispiel: Quicksort:**

**Aufgabe:** Ordne die Elemente in einem Array  $a[1..n]$  so, daß gilt:  $\forall i, j \in 1..n$ :  
 $i \leq j \implies a[i] \leq a[j]$ .

**Beobachtung:**

Existiert ein Index  $k$  mit  $k \in 2..n-1$  und  $\forall i \in 1..n$ ,  
 $i \leq k \implies a[i] \leq a[k]$  und  $\forall j \in 1..n$ ,  $k \leq j \implies a[k] \leq a[j]$ , dann reduziert sich die Aufgabe auf das Sortieren zweier kleinerer Arrays.



**Algorithmusgerüst:**

**Sortiere (a, i, j)**  
 Führe Trennung an  $a[k]$  durch  
 Sortiere (a, i, k-1)  
 Sortiere (a, k+1, j)

**Beispiel eines Trennbaums:**

**unsortiert:** Trennelemente sind kursiv und unterstrichen.

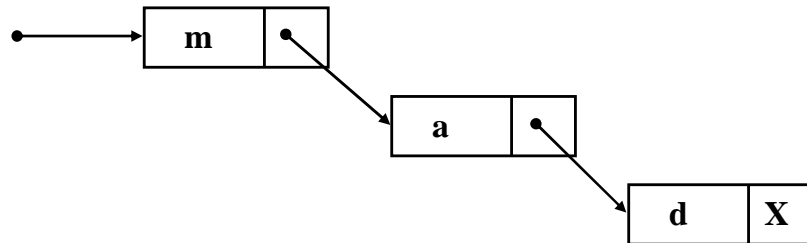
3 1 4 1 5 9 2 6 5 3  
 2 1 1 3 5 9 3 6 5 4  
 1 1 2 4 3 5 6 9 5  
1 1 3 4 5 6 9

**sortiert:**

1 1 2 3 3 4 5 5 6 9

## Lineare verkettete Liste:

Bild:



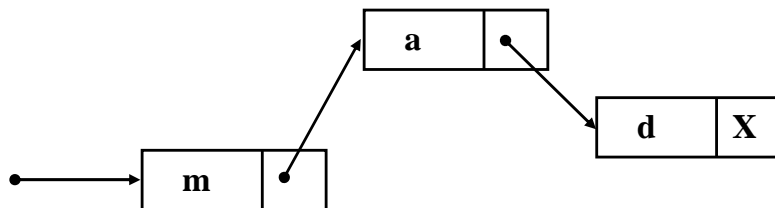
Viele Operationen auf linearen Listen lassen sich auf natürliche Weise rekursiv formulieren, z. B. das Kopieren einer Liste:

Kopiere erstes Element,  
Kopiere Restliste.

Hier Sonderfälle:

Kopieren eines Listenelementes,  
Kopieren einer leeren Liste.

Beispiel: Kopie obiger Liste:



## Lineare Rekursion:

Programme lassen sich auf vielfältige Art und Weise schreiben. So kann man oft zwischen einem iterativen und einem rekursiven Programmaufbau wählen. Die beiden folgenden Programmfragmente berechnen die Summe der Komponenten eines Arrays.

```
...  
int arr [] =           // Globaler Array  
  
const int anzahl = sizeof (arr) / sizeof (int);  
  
int RS (int i) {  
    if (i < anzahl)  
        return (arr [i] + RS (i+1));  
    else  
        return 0;  
} // rekursive Formulierung  
  
int IS () {  
    int sum = 0;  
    for (int i = 0; i < anzahl; ++i)  
        sum = sum + arr [i];  
    return (sum);  
} // iterative Formulierung
```

**Bemerkung:** Der Kernaufwand in den Prozeduren RS und IS ist gleich, der Verwaltungsaufwand verschieden.

### Schematransformation:

Sei  $p$  ein wohldefiniertes Prädikat, sei  $G$  eine wohldefinierte Funktion.

A:  $F(x) = \text{if } p(x) \text{ then } F(G(x))$

B:  $F(x) = \text{while } p(x) \text{ do}$   
     $x := G(x)$   
end

Betrachtet man nur Werteparameter, dann sind offensichtlich beide Darstellungen gleichwertig.

### Beispiel: GGT natürlicher Zahlen:

#### Rekursive Formulierung:

$\text{GGT}(x, y) = \text{if } y = 0 \text{ then}$   
     $x$   
else  
     $\text{GGT}(y, x \bmod y)$

#### Iterative Formulierung:

$\text{GGT}(x, y) = \text{while } y \neq 0 \text{ do}$   
    neu  $z := x \bmod y$   
     $x := y$   
     $y := z$   
end  
 $x$

### Beispiel: Potenz $x^y$ mit $y$ natürliche Zahl:

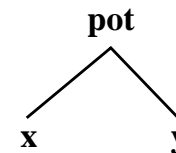
#### Rekursive Beschreibung:

$\text{pot}(x, y) = \text{if } y = 0 \text{ then}$   
     $1$   
else  
     $x * \text{pot}(x, y-1)$

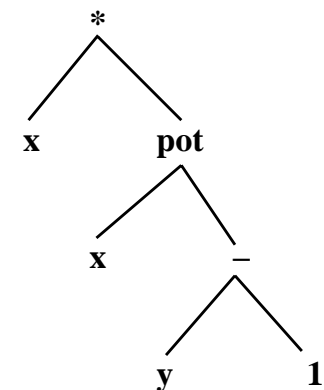
Obige Form genügt nicht dem Schema A, denn die äußere Funktion ist "\*" und nicht pot. Im Fall der linearen Rekursion läßt sich immer eine allgemeinere Funktion finden, die dem Schema A genügt. Ein mögliches Vorgehen benutzt Baumabgleiche.

#### Baumabgleich:

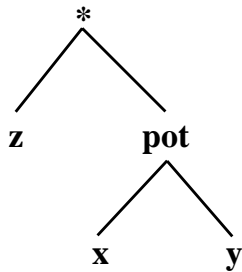
Linke Seite



Rechte Seite



Möglicher verallgemeinerter Ausdruck:



Nur die Baumspitzen müssen zur Deckung gebracht werden können, die Argumente der rekursiven Funktion stören nicht.

Für die verallgemeinerte Funktion  $g$  berechnet man:

$$\begin{aligned} g(z, x, y) &= z * \text{pot}(x, y) \\ &= z * (\text{if } y = 0 \text{ then } 1 \text{ else } x * \text{pot}(x, y - 1)) \\ &= \text{if } y = 0 \text{ then } 1 * z \text{ else } z * x * \text{pot}(x, y - 1) \\ &= \text{if } y = 0 \text{ then } z \text{ else } g(z * x, x, y - 1) \end{aligned}$$

Die Funktion  $g$  genügt nun dem Schema A. Eine Spezialisierung der Funktion  $g$  ist die Funktion  $\text{pot}$ , denn:

$$\begin{aligned} g(1, x, y) &= 1 * \text{pot}(x, y) \\ &= \text{pot}(x, y) \end{aligned}$$

Schleifenprogramm:

```
z := 1
while y ≠ 0 do
    z := z * x
    y := y - 1
end
z // Ergebnis ist der Wert von z
```

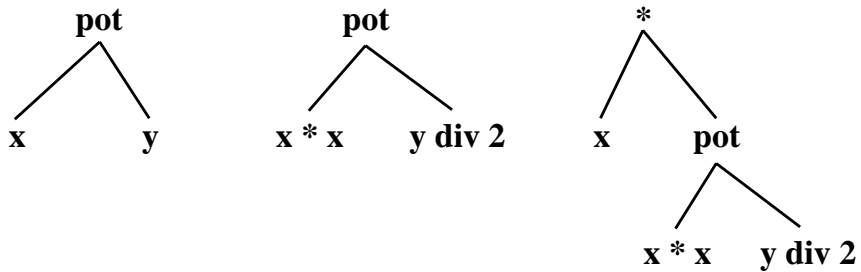
Rechenregeln für die Potenz  $x^y$ :

$$x^y = \begin{cases} 1 & y = 0 \\ (x^2)^{y \text{ div } 2} & y \neq 0 \wedge y \text{ gerade} \\ x * (x^2)^{y \text{ div } 2} & y \neq 0 \wedge y \text{ ungerade} \end{cases}$$

Dies führt sofort zur rekursiven Funktion:

```
pot(x, y) = if y = 0 then
    1
elseif even(y) then
    pot(x * x, y div 2)
else
    x * pot(x * x, y div 2)
```

### Baumabgleich:



Die verallgemeinerte Funktion ist wieder:

$$g(z, x, y) = z * \text{pot}(x, y)$$

Damit erhält man folgende Schleifenform für  $x^y$

```
z := 1;
while y ≠ 0 do
  if even(y) then
    x := x * x;
    y := y div 2
  else
    z := z * x;
    x := x * x;
    y := y div 2
end;
z
```

**Bemerkung:** Man vergleiche diese Herleitung mit einer Herleitung über Schleifeninvarianten.

**Ziele bei der Umformung eines rekursiven Programms in ein iteratives.**

1. **Bessere Platznutzung, oft erfolgreich.**
2. **Bessere Zeitnutzung, manchmal erfolgreich.**
3. **Gewinnung neuer Einsichten im zugrundeliegenden Algorithmus, Verbesserung des Algorithmus, unbestimmt erfolgreich.**

**Ziel 3 führt zu den Transformationszyklen:**

- ( $\alpha$ ) **rekursiv ==> iterativ ==> rekursiv**
- ( $\beta$ ) **iterativ ==> rekursiv ==> iterativ**