2.3 Merge-Sort

Sei wiederum eine Folge $a_1, \ldots, a_n$ von Schlüsseln im Feld $A[1..n]$ abgespeichert.

Algorithmus MergeSort

proc merge (r, s) **co** sortiere $A[r..s]$**oc**
  if $s \leq r$ **return fi**
  merge$(r, \lfloor \frac{r+s}{2} \rfloor)$; merge$(\lfloor \frac{r+s}{2} \rfloor + 1, s)$
  verschmelze die beiden sortierten Teilfolgen
end

merge$(1, n)$

Das Verschmelzen sortierter Teilfolgen $A[r..m]$ und $A[m+1, s]$ funktioniert wie folgt unter Benutzung eines Hilfsfeldes $B[r, s]$:

$i := r$; $j := m + 1$; $k := r$
while $i \leq m$ **and** $j \leq s$ **do**
  if $A[i] < A[j]$ **then** $B[k] := A[i]$; $i := i + 1$
  else $B[k] := A[j]$; $j := j + 1$ **fi**
  $k := k + 1$
od
if $i \leq m$ **then** kopiere $A[i, m]$ nach $B[k, s]$
else kopiere $A[j, s]$ nach $B[k, s]$ **fi**
kopiere $B[r, s]$ nach $A[r, s]$ zurück

Nach Beendigung von MergeSort ist das Feld A aufsteigend sortiert.

Satz 139

MergeSort *sortiert ein Feld der Länge* n *mit maximal* $n \cdot \lceil \mathrm{ld}(n) \rceil$ *Vergleichen.*

Beweis:

In jeder Rekursionstiefe werde der Vergleich dem kleineren Element zugeschlagen. Dann erhält jedes Element pro Rekursionstiefe höchstens einen Vergleich zugeschlagen.

2.4 Quick-Sort

Beim Quick-Sort-Verfahren wird in jeder Phase ein Element p der zu sortierenden Folge als Pivot-Element ausgewählt (wie dies geschehen kann, wird noch diskutiert). Dann wird in situ und mit einer linearen Anzahl von Vergleichen die zu sortierende Folge so umgeordnet, dass zuerst alle Elemente $< p$, dann p selbst und schließlich alle Elemente $> p$ kommen. Die beiden Teilfolgen links und rechts von p werden dann mit Quick-Sort rekursiv sortiert (Quick-Sort ist also ein Divide-and-Conquer-Verfahren).

Quick-Sort benötigt im schlechtesten Fall, nämlich wenn als Pivot-Element stets das kleinste oder größte der verbleibenden Elemente ausgewählt wird,

$$\sum_{i=1}^{n-1}(n-i) = \binom{n}{2}$$

Vergleiche.

Satz 140

QuickSort *benötigt zum Sortieren eines Feldes der Länge* n *durchschnittlich nur*

$$2\ln(2) \cdot n \operatorname{ld}(n) + O(n)$$

viele Vergleiche.

Beweis:

Siehe Vorlesung Diskrete Wahrscheinlichkeitstheorie (DWT).

Entscheidend für die Laufzeit von Quick-Sort ist eine „gute“ Wahl des Pivotelements. U.a. werden folgende Varianten verwendet:

1. Nimm stets das letzte Element der (Teil-)Folge als Pivotelement
   Nachteil: sehr schlecht bei vorsortierten Arrays!
2. Median-of-3 Verfahren: Wähle den Median (das mittlere Element) des ersten, mittleren und letzten Elements des Arrays
   Analyse Übungsaufgabe!
3. Wähle ein zufälliges Element als Pivotelement
   liefert die o.a. durchschnittliche Laufzeit, benötigt aber einen Zufallsgenerator.

2.5 Heap-Sort

Definition 141

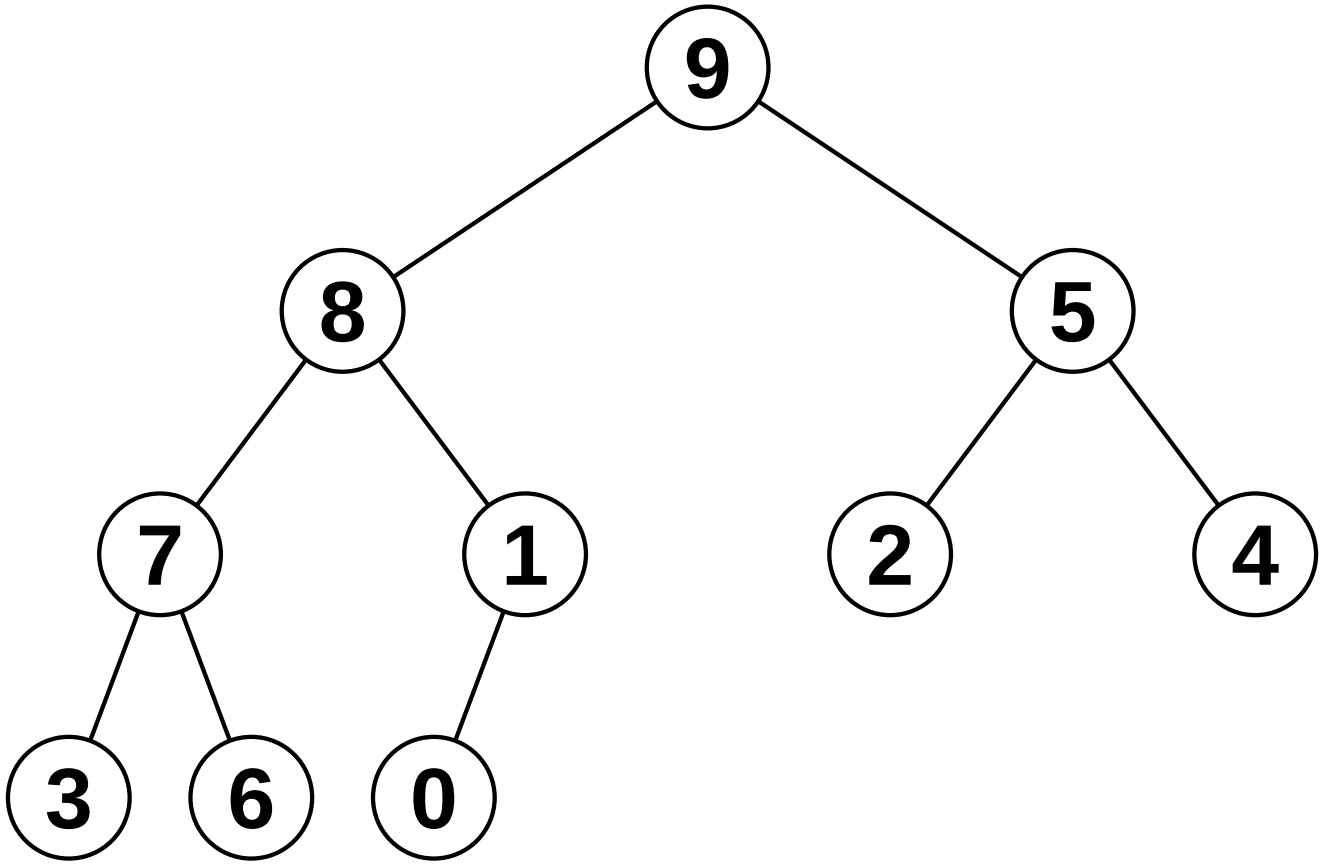
Ein Heap ist ein Binärbaum, an dessen Knoten Schlüssel gespeichert sind, so dass gilt:

1. alle inneren Knoten bis auf maximal einen haben genau zwei Kinder;
2. alle Knoten mit weniger als zwei Kindern (also insbesondere die Blätter) befinden sich auf der untersten oder der zweituntersten Schicht;
3. die unterste Schicht ist von links nach rechts aufgefüllt;
4. für jeden Knoten (mit Ausnahme der Wurzel) gilt, dass sein Schlüssel kleiner ist als der seines Vaters (Heap-Bedingung).

Bemerkungen:

1. Die hier definierte Variante ist ein max-Heap.
2. Die Bezeichnung heap wird in der Algorithmentheorie auch allgemeiner für Prioritätswarteschlangen benutzt!

Beispiel 142



Der Algorithmus HeapSort besteht aus zwei Phasen.

1. In der ersten Phase wird aus der unsortierten Folge von n Elementen ein Heap gemäß Definition aufgebaut.
2. In der zweiten Phase wird dieser Heap ausgegeben, d.h. ihm wird n-mal jeweils das größte Element entnommen (das ja an der Wurzel steht), dieses Element wird in die zu sortierende Folge aufgenommen und die Heap-Eigenschaften wird wieder hergestellt.

Betrachten wir nun zunächst den Algorithmus REHEAP zur Korrektur der Datenstruktur, falls die Heap-Bedingung höchstens an der Wurzel verletzt ist.

Algorithmus REHEAP

sei v die Wurzel des Heaps;
while Heap-Eigenschaft in v nicht erfüllt **do**
  sei v' das Kind von v mit dem größeren Schlüssel
  vertausche die Schlüssel in v und v'
  $v := v'$
od

Zweite Phase von HeapSort

for $i := n$ **downto** 1 do
  sei r die Wurzel des Heaps
  sei k der in r gespeicherte Schlüssel
  $A[i] := k$
  sei b das rechteste Blatt in der untersten Schicht des Heaps
  kopiere den Schlüssel von b in die Wurzel r
  entferne das Blatt b
  Reheap
od

Nach Beendigung der zweiten Phase von HeapSort ist das Feld $A[1..n]$ aufsteigend sortiert.

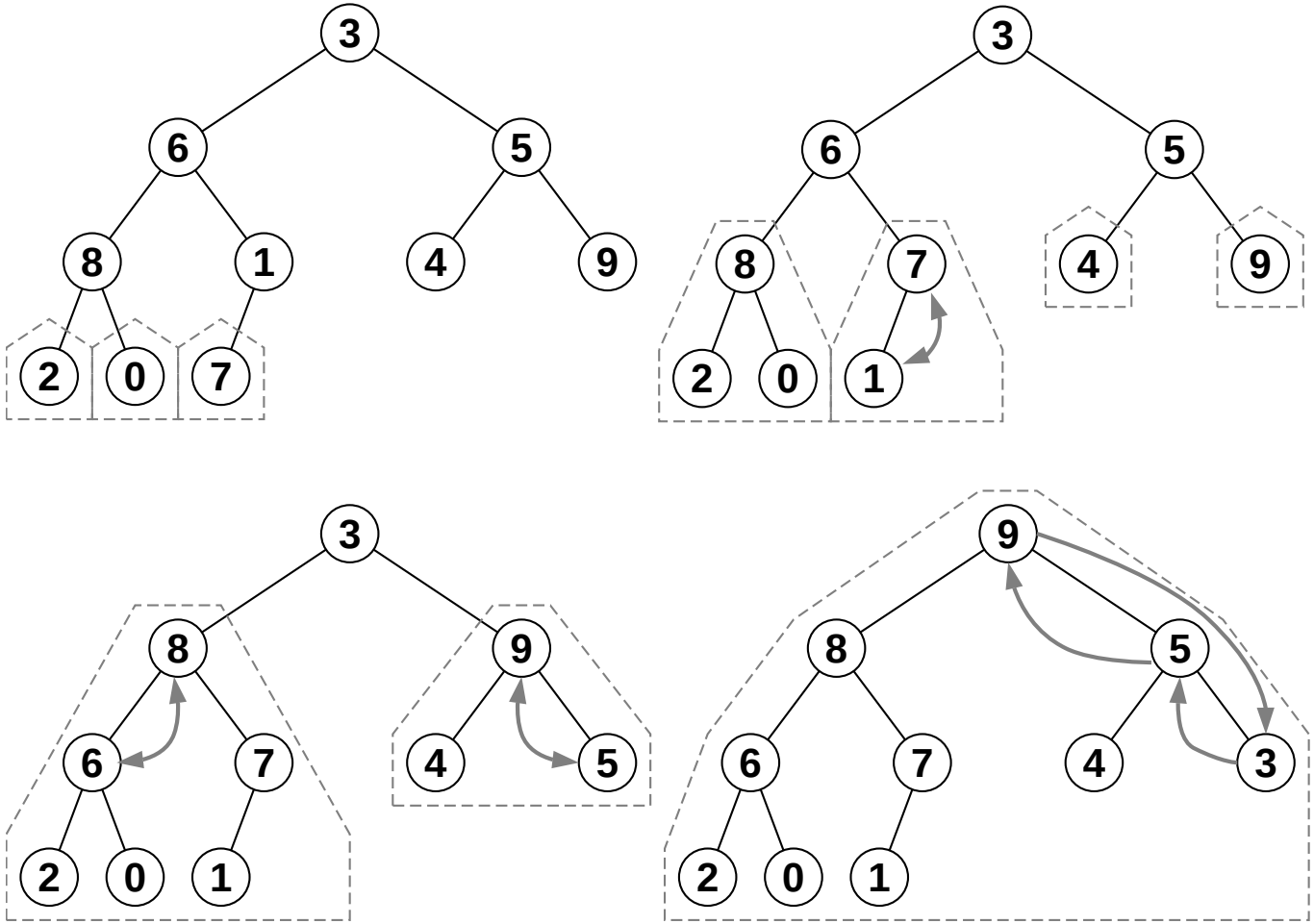
Mit geeigneten (kleinen) Modifikationen kann HeapSort in situ implementiert werden. Die Schichten des Heaps werden dabei von oben nach unten und von links nach rechts im Feld A abgespeichert.

In der ersten Phase von HeapSort müssen wir mit den gegebenen n Schlüsseln einen Heap erzeugen.

Wir tun dies iterativ, indem wir aus zwei bereits erzeugten Heaps und einem weiteren Schlüssel einen neuen Heap formen, indem wir einen neuen Knoten erzeugen, der die Wurzel des neuen Heaps wird. Diesem neuen Knoten ordnen wir zunächst den zusätzlichen Schlüssel zu und und machen die beiden alten Heaps zu seinen Unterbäumen. Damit ist die Heap-Bedingung höchstens an der Wurzel des neuen Baums verletzt, was wir durch Ausführung der Reheap-Operation korrigieren können.

Beispiel 143

[Initialisierung des Heaps]



Lemma 144

Die Reheap-Operation erfordert höchstens $O($Tiefe des Heaps$)$ Schritte.

Beweis:

Reheap führt pro Schicht des Heaps nur konstant viele Schritte aus.

Lemma 145

Die Initialisierung des Heaps in der ersten Phase von HeapSort *benötigt nur $O(n)$ Schritte.*

Beweis:

Sei d die Tiefe (Anzahl der Schichten) des (n-elementigen) Heaps. Die Anzahl der Knoten in Tiefe i ist $\leq 2^i$ (die Wurzel habe Tiefe 0). Wenn ein solcher Knoten beim inkrementellen Aufbau des Heaps als Wurzel hinzugefügt wird, erfordert die Reheap-Operation $\leq d - i$ Schritte, insgesamt werden also

$$\leq \sum_{i=0}^{d-1} (d-i)2^i = O(n)$$

Schritte benötigt.

Satz 146

HeapSort *benötigt maximal* $O(n \log n)$ *Schritte.*

Beweis:

In der zweiten Phase werden $< n$ Reheap-Operationen auf Heaps der Tiefe $\leq \log n$ durchgeführt.

Bemerkung:

1. Eine genauere Analyse ergibt eine Schranke von $2n \operatorname{ld}(n) + o(n)$.
2. Carlsson hat eine Variante von HeapSort beschrieben, die mit $n \operatorname{ld} n + O(n \log \log n)$ Vergleichen auskommt:

   Svante Carlsson:
   A variant of heapsort with almost optimal number of comparisons.
   Inf. Process. Lett., **24**(4):247–250, 1987

2.6 Vergleichsbasierte Sortierverfahren

Alle bisher betrachteten Sortierverfahren sind vergleichsbasiert, d.h. sie greifen auf Schlüssel k, k' (außer in Zuweisungen) nur in Vergleichsoperationen der Form $k < k'$ zu, verwenden aber nicht, dass etwa $k \in \mathbb{N}_0$ oder dass $k \in \Sigma^*$.

Satz 147

Jedes vergleichsbasierte Sortierverfahren benötigt im worst-case mindestens

$$n \operatorname{ld} n + O(n)$$

Vergleiche und hat damit Laufzeit $\Omega(n \log n)$.

Beweis:

Wir benutzen ein so genanntes Gegenspielerargument (engl. adversary argument). Soll der Algorithmus n Schlüssel sortieren, legt der Gegenspieler den Wert eines jeden Schlüssels immer erst dann fest, wenn der Algorithmus das erste Mal auf ihn in einem Vergleich zugreift. Er bestimmt den Wert des Schlüssels so, dass der Algorithmus möglichst viele Vergleiche durchführen muss.

Am Anfang (vor der ersten Vergleichsoperation des Algorithmus) sind alle $n!$ Sortierungen der Schlüssel möglich, da der Gegenspieler jedem Schlüssel noch einen beliebigen Wert zuweisen kann.

Beweis:

Seien nun induktiv vor einer Vergleichsoperation $A[i] < A[j]$ des Algorithmus noch r Sortierungen der Schlüssel möglich.

Falls der Gegenspieler die Werte der in $A[i]$ bzw. $A[j]$ gespeicherten Schlüssel bereits früher festgelegt hat, ändert sich die Anzahl der möglichen Sortierungen durch den Vergleich nicht, dieser ist redundant.

Beweis:

Andernfalls kann der Gegenspieler einen oder beide Schlüssel so festlegen, dass immer noch mindestens $r/2$ Sortierungen möglich sind (wir verwenden hier, dass die Schlüssel stets paarweise verschieden sind).

Nach k Vergleichen des Algorithmus sind also immer noch $n!/2^k$ Sortierungen möglich. Der Algorithmus muss jedoch Vergleiche ausführen, bis nur noch eine Sortierung möglich ist (die dann die Ausgabe des Sortieralgorithmus darstellt).

Damit

$$\#\text{Vergleiche} \geq \lceil \mathrm{ld}(n!) \rceil = n \,\mathrm{ld}\, n + O(n)$$

mit Hilfe der Stirlingschen Approximation für $n!$. □

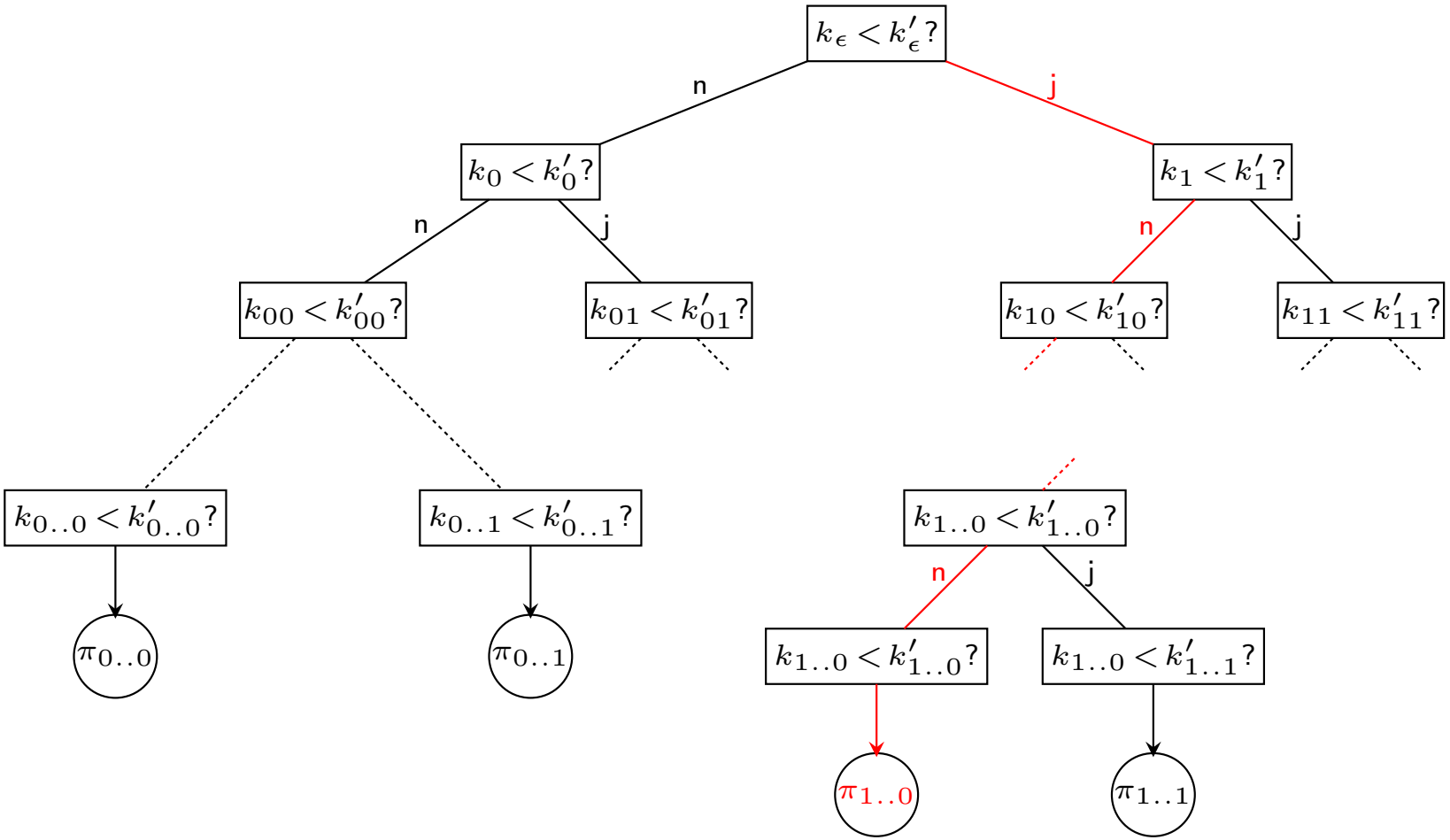
Alternativer Beweis mit Hilfe des Entscheidungsbaums

Ein vergleichsbasierter Algorithmus kann auf die Werte der Schlüssel nur durch Vergleiche $k < k'$ zugreifen. Wenn wir annehmen, dass alle Schlüssel paarweise verschieden sind, ergeben zu jedem Zeitpunkt die vom Algorithmus erhaltenen binären Antworten auf seine Anfragen der obigen Art an die Schlüsselmenge seine gesamte Information über die (tatsächliche) Ordnung der Schlüssel.

Am Ende eines jeden Ablaufs des Algorithmus muss diese Information so sein, dass die tatsächliche Ordnung der Schlüssel eindeutig festliegt, dass also nur eine Permutation der n zu sortierenden Schlüssel mit der erhaltenen Folge von Binärantworten konsistent ist.

Wir stellen alle möglichen Abläufe (d.h., Folgen von Vergleichen), die sich bei der Eingabe von n Schlüsseln ergeben können, in einem so genannten Entscheidungsbaum dar.

Entscheidungsbaum:



Damit muss es für jede der $n!$ Permutationen mindestens ein Blatt in diesem Entscheidungsbaum geben. Da dieser ein Binärbaum ist, folgt daraus:

- Die Tiefe des Entscheidungsbaums (und damit die Anzahl der vom Sortieralgorithmus benötigten Vergleiche im worst-case) ist
$$\geq \lceil \mathrm{ld}(n!) \rceil = n \,\mathrm{ld}\, n + O(n)\,.$$
- Diese untere Schranke gilt sogar im Durchschnitt (über alle Permutationen).