Kombinatorische Algorithmen in C

Axel Kohnert
24.7.1997

Permutationen I

Permutationen sind Bijektionen zwischen endlichen Mengen. Dazu kann man o.E. die Menge {1,2,3,...,n} =: $\underline{n}$ der Zahlen 1 bis n als endliche Menge nehmen. Eine Bijektion π, ist dann durch die Bilder der einzelnen Zahlen gegeben. Die Menge der Permutationen einer n-elementigen Menge wird mit S_n bezeichnet. Die Zahl n wird als *Grad* der Permutation bezeichnet. Eine Bijektion zwischen einer 4-elementigen Menge ist gegeben durch die Festlegung

$$\pi(1) = 2, \pi(2) = 3, \pi(3) = 4, \pi(4) = 1.$$

Hierfür wählt man auch die Notation

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{bmatrix}$$

oder aber noch kürzer

$$\begin{bmatrix} 2 & 3 & 4 & 1 \end{bmatrix}$$

wobei man die obere Zeile wegläßt. Diese Notation bezeichnet man als *Listenschreibweise*.Um den Unterschied klarzumachen sagt man manchmal auch lange, b.z.w. verkürzte Listenschreibweise. Punkt die unter π auf sich selber abgebildet werden, heißen *Fixpunkte* der Permutation.

Permutationen in Symmetrica

In Symmetrica gibt es die Objektklasse PERMUTATION, dabei werden die Permutationen in verkürzter Listenschreibweise dargestellt. Folgendes Programm liest ein PERMUTATION Objekt ein, und gibt es wieder aus:

source code	Eingabe	Ausgabe

`/* SYMMETRICA p1.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(PERMUTATION,a);`
`println(a);`
`ENDE`
	`4 2 3 4 1`	`....`
		`[2,3,4,1]`
		`....`

Bei der Ausgabe wurden einige Steuermitteilung des Programms unterdrückt. Bei der Eingabe muß zuerst der Grad der Permutation eingegeben werden.

symmetrische Gruppe

Die Menge S_n der Permutationen von festen Grad n bilden eine Gruppe, die *symmetrische Gruppe* vom Grad n. Die Verknüpfung in der Gruppe ist die Hintereinanderausführung der Bijektionen, sie wird als Multiplikation der Permutationen bezeichnet. Das neutrale Element ist die Permutation [1,2,...,n]. Das Produkt $\begin{bmatrix} 2 & 3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 2 & 4 & 1 \end{bmatrix}$ ist z.B. $\begin{bmatrix} 4 & 3 & 1 & 2 \end{bmatrix}$ was sich so ergibt:

$$\begin{bmatrix} 1 \rightarrow 3 \rightarrow 4 \\ 2 \rightarrow 2 \rightarrow 3 \\ 3 \rightarrow 4 \rightarrow 1 \\ 4 \rightarrow 1 \rightarrow 2 \end{bmatrix}$$

D.h. zuerst wird die rechts stehende Permutation angewandt. An dieser Stelle muß man aufpassen, da es auch andere Konventionen gibt. Die symmetrische Gruppe ist nicht kommutativ, sonst müsste man auch nicht auf die Reihenfolge der Multiplikation aufpassen. Die inverse Permutation erhält man indem man in der langen Listenschreibweise die beiden Zeilen vertauscht, und dann die Spalten gemäß der oberen Zeile sortiert:

$$[2, 3, 4, 1] \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 3 & 4 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{bmatrix} \rightarrow [4, 1, 2, 3]$$

Die Ordnung der symmetrischen Gruppe ist n!.

Die symmetrische Gruppe in Symmetrica

Die Multiplikation und das Invertieren stehen als Funktionen in Symmetrica zur Verfügung:

source code	Eingabe	Ausgabe

`/* SYMMETRICA p2.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(PERMUTATION,a);`
`scan(PERMUTATION,b);`
`mult(a,b,c);`
`println(c);`
`invers(a,d);`
`println(d);`
`ENDE`
	`4 2 3 4 1`
	
	`4 3 2 4 1`
		`....`
		`[4,3,1,2]`
		`[4,1,2,3]`
		`....`

Zykelschreibweise

Man kann jede Permutation in *Zykel* zerlegen. Ein Zykel der Länge k ist eine Permutation π der Form: die Permutation hat n-k Fixpunkte, die Element $i_1,...,i_k$ sind nicht Fixpunkte und unter π geht i_1 auf i_2, i_2 auf i_3, ... , i_{k-1} auf i_k und i_k auf i_1. Unser obiges Beispiel $\begin{bmatrix} 2 & 3 & 4 & 1 \end{bmatrix}$ war ein Zykel der Länge 4. Fixpunkt sind per Definition Zykel der Länge 1. Disjunkte Zykel sind Zykel deren Nichtfixpunkte disjunkt sind. Disjunkte Zykel kommutieren. Jede Permutation kann als Produkt disjunkter Zykel geschrieben werden.

$$[2, 5, 4, 3, 1] = [2, 5, 3, 4, 1] \cdot [1, 2, 4, 3, 5]$$

Für Zykel gibt es die Zykelschreibweise, man notiert die Punkte $i_1, ... , i_k$ in runden Klammern $(i_1, ... , i_k)$. Jede Permutation kann dann als Produkt von disjunkten Zykeln geschrieben werden.

$$[2, 5, 4, 3, 1] = (1, 2, 5)(4, 3) = (5, 1, 2)(3, 4)$$

Diese Zerlegung in Zykel ist nicht eindeutig, da man einen Zykel mit jedem Element innerhalb des Zykels beginnen kann. Man kann diese Schreibweise eindeutig machen indem man innerhalb der Zykel mit dem kleinsten Element beginnt, und die Zykel absteigend nach dem kleinsten Elementen sortiert.

$$[2, 5, 4, 3, 1] = (3, 4)(1, 2, 5)$$

Diese Notation erlaubt es sogar die runden Klammern zu streichen, und die richtige Zykelschreibweise kann aus der Zahlenfolge rekonstruiert werden. Dies liefert eine Bijektion innerhalb der S_n . Eine weitere Bijektion erhält man, wenn man innerhalb des Zykels das größte Element nach vorne stellt, und die Zykel dann ansteigend sortiert.

Invertieren in Zykelschreibweise

Das geht ganz einfach, man liest die Zyklen rückwärts. In unserem Beispiel:

Die Zykelschreibweise in Symmetrica

Diese Bijektion wird in Symmetrica verwendet, um auch die Zykelschreibweise platzsparend zu implementieren. Es gibt Funktionen um zwischen den beiden Notationen für Permutationen zu wechseln.

source code	Eingabe	Ausgabe

`/* SYMMETRICA p3.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(PERMUTATION,a);`
`t_VECTOR_ZYKEL(a,b);`
`println(b);`
`t_ZYKEL_VECTOR(b,c);`
`println(c);`
`ENDE`
	`5 2 5 4 3 1`	`....`
		`(3,4)(1,2,5)`
		`[2,5,4,3,1]`
		`....`

Invertieren einer Permutation ohne Hilfsfeld

Die oben vorgestellte Methode zur Berechnung der Inversen einer Permutation in Listenschreibweise benötigt ein Hilfsfeld. Der folgende Algorithmus (nach Knuth) kann dies ohne Hilfsfeld dazu wird die Permutation zykelweise abgearbeitet.

EINGABE:	π(1),....,π(n)
AUSGABE:	π(1),....,π(n) die inverse Permutation

ALGORITHMUS:	I1: m:=n
	
	I2: [nächste Element] i:=π(m).
	
	if (i<0) {π(m) = -i ; goto I6.}
	[dieses Element wurde bereits bearbeitet)
	
	if (i==m) goto I6 [dieses Element ist ein Fixpunkt]
	
	I3: k:=m
	
	I4: [π(k) = i in der Eingabe] j:= π(i); π(i):= -k;
	
	I5: [Zykel zu Ende ?]
	
	if (j==m) π(m):= i; [ja]
	
	else { k:=i; i:=j; goto I4; } [nein]
	
	I6:m--; if (m>0) goto I2;
	
	ENDE

Dazu ein Beispiel, wir wollen die Permutation $[2, 5, 4, 3, 1]$ am Platz invertieren. Als erstes wird der Zykel betrachtet, der am letzten Platz 5 endet. Von dort aus geht man zum Platz 1, dort kommt die -5 hin, von dort zum Platz 2, dort kommt die -1 hin und von dort wieder zurück zum Platz 5, wo die 2 hinkommt. Der erste Zykel ist beendet, die Permutation hat die Form $[-5, -1, 4, 3, 2]$. Der Eintrag am Platz 5 hat bereits die endgültige Form. Nun wird der Zykel zum Platz 4 betrachtet, und nachdem dieser 2-er Zykel beendet ist schaut die Permutation so aus: . Im restlichen Teil des Algorithmus werden nur noch die Minuszeichen entfernt und man erhält das richtige Ergebnis $[5, 1, 4, 3, 2]$

Transpositionen

Zykel der Länge 2 werden als *Transpositionen* bezeichnet. Dies sind Permutationen die genau zwei Elemente vertauschen und der Rest sind Fixpunkte. Man kann jede Permutation als Produkt von Transpositionen schreiben, d.h. Transpositionen sind ein Erzeuger der symmetrischen Gruppe. Eine Methode dies zu sehen ist ein Algorithmus die gegebene Permutation aus der Identität herzuleiten indem man nur Transpositionen verwendet. Gegeben sei die Permutation $[2, 5, 4, 3, 1]$ mittels Transpositionen werden nun die Ziffern an die richtigen Plätze gebracht:

$$\begin{bmatrix}2\\5\\4\\3\\1\end{bmatrix} \leftarrow (3{,}4) \leftarrow \begin{bmatrix}2\\5\\3\\4\\1\end{bmatrix} \leftarrow (1{,}5) \leftarrow \begin{bmatrix}2\\1\\3\\4\\5\end{bmatrix} \leftarrow (1{,}2) \leftarrow \begin{bmatrix}1\\2\\3\\4\\5\end{bmatrix}$$

So erhält man die Permutation $[2, 5, 4, 3, 1]$ als das Produkt $(3, 4)(1, 5)(1, 2)$, diese Schreibweise ist nicht eindeutig, ebenso ginge $(2, 5)(3, 4)(1, 5)$, wenn man zuerst die letzte Ziffer an den richtigen Platz bringt.

Elementartranspositionen

Transpositionen von benachbarten Ziffern werden *Elementartranspositionen* genannt. Davon gibt es in der S_n n-1 Stück, die üblicherweise mit $\sigma_1 = (1, 2), \ldots, \sigma_{n-1} = (n-1, n)$ bezeichnet werden. Bereits die Elementartranspositionen erzeugen die symmetrische Gruppe, man sieht dies indem man eine beliebige Transposition als Produkt von Elementartranspositionen schreibt. Sei (i, j) eine Transposition mit $i < j$, dann ist $(i, j) = \sigma_{j-1}\sigma_{j-2}\ldots\sigma_{i+1}\sigma_i\sigma_{i+1}\ldots\sigma_{j-1}$, d.h. für eine Transposition (i, j) benötigt man $2|i-j|-1$ Elementartranspositionen. Dies ist auch die minimale Anzahl. Man bemerkt ferner, daß das Produkt der Elementartranspositionen symmetrisch zur Mitte sein muß da jede Transposition zu sich selbst invers ist. Allgemein heißen Permutationen mit dieser Eigenschaft *Involutionen*, da zweimal nacheinander angewandt sie die Identität ergeben.

Elementartranspositionen in Symmetrica

In Symmetrica gibt es eine Routine, die eine Permutation als Produkt von Elementartranspositionen schreibt.

source code	Eingabe	Ausgabe

`/* SYMMETRICA p4.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(PERMUTATION,a);`
`rz(a,b);`
`println(b);`
`ENDE`
	`5 5 2 3 4 1`	`....`
		`[4,3,2,1,2,3,4]`
		`....`

Die Routine rz erzeugt einen Vektor, dessen Einträge die Indizes der Elementartranspositionen sind.

Inversionen und Inversionstabellen

Damit meint man Fehlstellungen, d.h. sucht Paare (i, j) mit $i < j$ und $\pi(i) > \pi(j)$. Ein solches Paar heißt *Inversion*. Um Inversionen leicht zu zählen verwendet man *Inversionstabellen*. Dies sind Vektoren der Länge n, die Permutationen aus S_n über ihre Inversionen kodieren. Hierfür gibt es mehrere Möglichkeiten.

Lehmercode

Dies ist eine Inversionstabelle, eine Abbildung $L:S_n \to N^n$. Der Wert $L(\pi)_i$ wird definiert als die Anzahl der $\pi(j)$ mit $j>i$ und $\pi(j) < \pi(i)$. Diese Inversionstabelle heißt *Lehmercode* von π. Die Anzahl der Inversionen ist dann die Summe über die Inversionstabelle. Zur Berechnung des Lehmercodes ein Beispiel. Der Lehmercode zur Permutation $[2, 5, 4, 3, 1]$ soll berechnet werden.

$\pi(i)$	kleinere rechts	L_i
2	1	1
5	4,3,1	3
4	3,1	2
3	1	1
1		0

Der Lehmercode ist also $[1, 3, 2, 1, 0]$. Aus der Konstruktion des Lehmercodes ist es klar, daß der i-te Eintrag kleiner gleich n-i ist. Es gibt genau n! solche Vektoren der Länge n, und man sieht leicht, daß der Lehmercode eine Bijektion zwischen diesen Vektoren und der S_n ist. Dazu gibt man am einfachsten eine Regel an die inverse Abbildung zu konstruieren. Dazu baut man den Lehmercode von links ab, und greift die L_1+1 -te Zahl aus den Zahlen 1...n. Dies ist der erste Eintrag in der Permutation. Nun greift man die L_2+1 -te Zahl aus den verbliebenen u.s.w. Es wird klar im folgenden Beispiel

Lehmercode	Zahlen	Permutation
1,3,2,1,0	1,2,3,4,5	-,-,-,-,-
-,3,2,1,0	1,3,4,5	2,-,-,-,-
-,-,2,1,0	1,3,4	2,5,-,-,-
-,-,-,1,0	1,3	2,5,4,-,-
-,-,-,-,0	1	2,5,4,3,-
-,-,-,-,-		2,5,4,3,1

Lehmercode in Symmetrica

Auch für diese Bijektion gibt es eine Routine in Symmetrica. Wird die Routine lehmercode auf ein PERMUTATION Objekt angewandt, so wird ein VECTOR Objekt errechnet, umgekehrt kann man mit der gleichen Routine aus einem VECTOR Objekt ein PERMUTATION Objekt errechnen.

source code	Eingabe	Ausgabe

`/* SYMMETRICA p5.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(PERMUTATION,a);`
`lehmercode(a,b);`
`println(b);`
`lehmercode(b,c);`
`println(c);`
`ENDE`
	`5 2 5 4 3 1`	`....`
		`[1,3,2,1,0]`
		`[2,5,4,3,1]`
		`....`

rank und unrank

rank und *unrank* Funktionen sind zueinander inverse Bijektionen zwischen einer Menge X und den Zahlen 1 bis $|X|$. Im Falle der Permutationen der S_n ist die rank Funktion eine Bijektion nach 1,...,n! und die unrank Funktion eine Bijektion der Zahlen 1,...,n! nach S_n . Bei der Konstruktion dieser Bijektion und ihrer Inversen ist der Lehmercode sehr nützlich. Dies war auch die ursprüngliche Motivation von D.N. Lehmer[1]. Man bemerkt, daß sich jede Zahl m zwischen 0 und n!-1 eindeutig schreiben läßt als:

$$m = \sum_{i=0}^{n-1} m_i i!$$

mit $m_i \le i$. Nimmt man als Beispiel die Zahl $19 < 4!$, dann erhält man die Darstellung

$$19 = 0 \cdot 0! + 1 \cdot 1! + 0 \cdot 2! + 3 \cdot 3!$$

Betrachtet man nun den Lehmercode einer Permutation aus der S_n, so ist dies ein Zahlenfolge, deren erster Eintrag $<n$, der zweite $<n-1$, usw ist, d.h. dies sind gerade die Koeffizienten m_i. D.h. im obigen Beispiel mit der $19 < 4!$ haben wir den Lehmercode 3,0,1,0. Dies ist die Permutation [4,1,3,2]. Also die Permutation wird unter der rank Funktion auf die Zahl 19 abgebildet. Bei Berechnung der unrank Funktion sucht man die Zerlegung der Zahl m, dies geschieht indem man schrittweise die Vielfachen der fallenden Fakultäten abzieht. Diese rank und unrank Funktionen sind eine Bijektion zwischen S_n und 0,...,n!-1.

weiteres ranking

Bei dem oben vorgestellten ranking wurde eine Zahl m zwischen 0 und n!-1 geschrieben als

$$m = \sum_{i=0}^{n-1} m_i c_i$$

mit $m_i \le i$. Dabei war $c_i = i!$. Mittels dieser Darstellung konnte dann mittels Lehmercode die Ranking Funktion definiert werdem. Die gute Nachricht ist: es gibt eine weitere Möglichkeiten c_i zu wählen, und so erhält man neue ranking Funktionen. Dies geschieht mittels

$$c_i = \frac{n!}{i!}$$

Will man die Zahl 19 so darstellen, so erhält man die Zerlegung:

$$19 = 0 \cdot 24 + 1 \cdot 12 + 1 \cdot 4 + 3 \cdot 1$$

Was den Lehmercode (3,1,1,0) b.z.w. die Permutation [4,2,3,1] liefert. Die Zahlen c_i werden als *fallende Faktorielle* bezeichnet und üblicherweise definiert als:

$$(n)_k = n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$$

1. Lehmer, D.N., *On the orderly listing of substitutions*, Bull. AMS **12** (1906), 81-84

rank und unrank in Symmetrica

Diese beiden Funktionen sind auch in Symmetrica implementiert. Da n! schnell ansteigt, wird schon ab n=13 eine Langzahlarithmetik benötigt.

source code	Eingabe	Ausgabe

`/* SYMMETRICA p6.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(PERMUTATION,a);`
`rank_permutation(a,b);`
`println(b);`
`unrank_degree_permutation(b,`
`S_P_L(a),c);`
`println(c);`
`ENDE`
	`15`
	`15 13 14 11 12 9`
	`10 7 8 5 6 3 4 1 2`
		`....`
		`1.301407.062352`
		`[15,13,14,11,12,9,10,7,`
		`8,5,6,3,4,1,2]`
		`....`

next

Die nächste Aufgabe, die sich stellt, wenn man mit kombinatorischen Objekten arbeitet, ist das Durchlaufen dieser Objekte. Hat man eine rank/unrank Funktion, so ist dies eine leichte Aufgabe. Will man die Permutationen der S_n durchlaufen, so beginnt man mit 0, ruft die Funktion unrank auf, und erhält so die erste Permutation, um die nächste zu erzeugen, wird mit rank die Nummer errechnet, diese um 1 erhöht, ist dies n! so hatte man bereits die letzte, ansonsten wieder unrank und man hat die nächste.

rank	Lehmercode	Permutation
0	[0,0,0]	[1,2,3]
1	[0,1,0]	[1,3,2]
2	[1,0,0]	[2,1,3]
3	[1,1,0]	[2,3,1]
4	[2,0,0]	[3,1,2]
5	[2,1,0]	[3,2,1]

Eine wichtige Eigenschaft dieses Algorithmus die Permutationen/ den Lehmercode zu durchlaufen ist, daß dies in *lexikographischer* Reihenfolge passiert. Die beschriebene Methode die lexikographisch nächste Permutation zu finden ist sehr aufwendig, man kann dies auch direkt implementieren:

EINGABE:	$\pi(1),....,\pi(n)$
AUSGABE:	$\pi(1),....,\pi(n)$ die lexikographisch nächste Permutation
ALGORITHMUS:	N1: i = rechteste Index mit $\pi(i) < \pi(i+1)$ [rechts von i ist die Permutation fallend] N2: j = Index des rechteste Elements rechts von i, welches größer ist als $\pi(i)$ N3: Vertausche $\pi(i)$ und $\pi(j)$ N4: Schreibe die Elemente $\pi(i+1)$ .... $\pi(n)$ in umgekehrter Reihenfolge ENDE

Hierzu ein Beispiel. Was ist die nächste Permutation nach [3, 6, 5, 8, 7, 4, 2, 1]. Im Schritt N1 wird i bestimmt, in diesem Beispiel ist i = 3. Nun wird rechts das kleinste Element gesucht, welches größer ist als 5, dies ist die 7 am Platz j=6. In N3 werden die beiden vertauscht, und die Permutation hat die Form [3, 6, 7, 8, 5, 4, 2, 1] und zuletzt wird in Schritt N4, der zuvor absteigende Teil umgedreht, und die lexikographisch nächste Permutation ist [3, 6, 7, 1, 2, 4, 5, 8] Dieser Algorithmus wurde bereits 1812 von L.J. Fischer und K.Chr. Krause vorgeschlagen. Dieser Algorithmus funktioniert auch mit Einträgen, die nicht unbedingt verschieden sind.

Permutation	i,j
11223	4,5
11232	3,4
11322	2,5
12123	4,5
12132	3,5
12213	....

next in Symmetrica

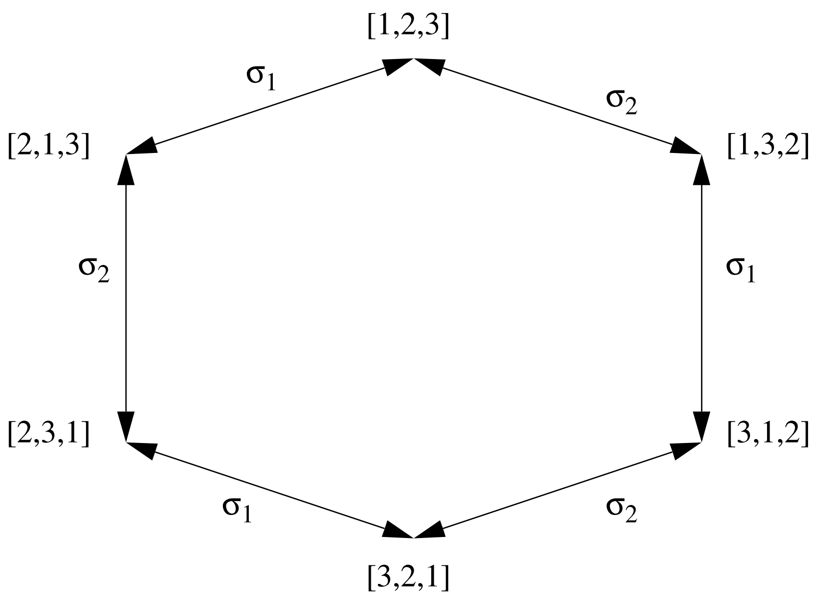
Ein Beispiel zur Verwendung von next in Symmetrica:

source code	Eingabe	Ausgabe

`/* SYMMETRICA p7.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(INTEGER,a);`
`first_permutation(a,b);`
`do {`
`        println(b);`
`} while (next(b,b));`
`ENDE`
	`3`	`....`
		`[1,2,3]`
		`[1,3,2]`
		`[2,1,3]`
		`[2,3,1]`
		`[3,1,2]`
		`[3,2,1]`
		`....`

Durchlaufen der Permutationen mittels Erzeuger

Wir haben gesehen, daß die Gruppe S_n durch die Elementartranspositionen erzeugt wird. Man kann dies durch den *Cayleygraphen* visualisieren. Die Knoten sind die Elemente der Gruppe, und eine gerichtete Kante (u,v) wird eingetragen, wenn man von u nach v mittels Multiplikation mit einem Erzeuger kommt. Man muß dazu noch festlegen ob es sich um Links- oder Rechtsmultiplikation handelt. Im Falle der symmetrischen Gruppe hat man ungerichtete Kanten da die Transpositionen Involutionen sind. Das Beispiel ist die symmetrische Gruppe vom Grad 3 und Rechtsmultiplikation.



Die Frage nun ist, gibt es einen *Hamiltonweg* in diesem Graphen, d.h. ein Weg durch alle Knoten ohne einen Knoten zweimal zu besuchen. Dies ist eine allgemeine Fragestellung bei Gruppen, die durch ihre Erzeuger gegeben sind.

Algorithmus von Trotter

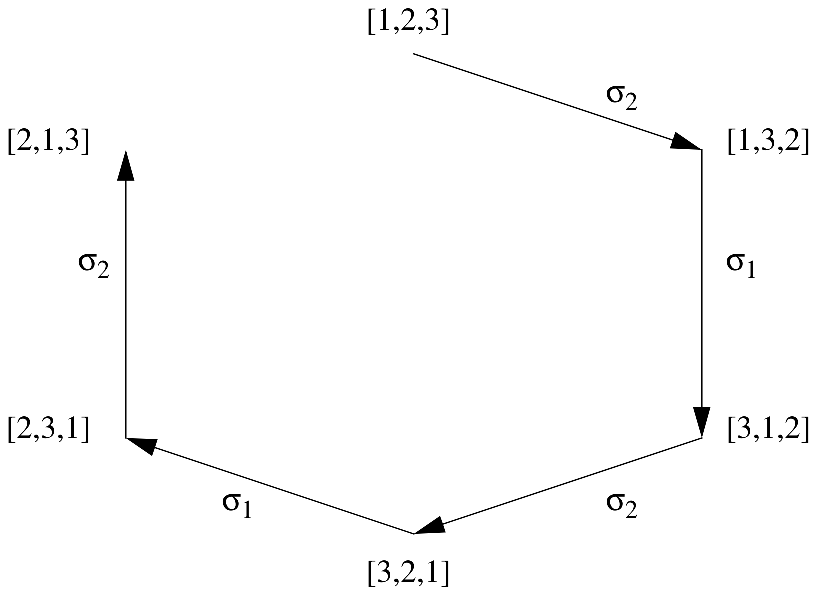
Dieser *Algorithmus von Trotter* (1962) erlaubt das Durchlaufen aller Permutation der S_n, wobei sich benachbarte Permutationen durch eine Elementartransposition unterscheidet. Um diesen Algorithmus zu formulieren erweitert man die Listenschreibweise, indem man bei den Permutationselementen eine Bewegungsrichtung mittels Pfeil notiert. Die Bewegungsrichtung zeigt entweder nach links oder nach rechts. Der Algorithmus startet mit dem Element $[1^{\leftarrow},2^{\leftarrow},3^{\leftarrow},...,n^{\leftarrow}]$. Die erste Permutation, ist die gleiche wie bei dem lexikographischen Durchlauf, zusätzlich ist die Bewegungsrichtung nach links bei allen Permutationselementen. Ein Element k der Permutation heißt *beweglich*, wenn in Bewegungsrichtung ein kleineres Element daneben steht.

EINGABE:	$\pi(1),....,\pi(n)$ Permutation mit Bewegungsrichtungen
AUSGABE:	$\pi(1),....,\pi(n)$ die nächste Permutation
ALGORITHMUS:	T1: existiert kein bewegliches Element, dann ENDE T2: sei m die größte bewegliche Zahl T3: tausche m mit dem Nachbarn in Bewegungsrichtung T4: ändere die Bewegungsrichtung für alle k>m

Hierzu ein Beispiel:

Permutation	m	Permutation	m
$[1^{\leftarrow},2^{\leftarrow},3^{\leftarrow},4^{\leftarrow}]$	4	$[4^{\rightarrow},3^{\rightarrow},2^{\leftarrow},1^{\leftarrow}]$	4
$[1^{\leftarrow},2^{\leftarrow},4^{\leftarrow},3^{\leftarrow}]$	4	$[3^{\rightarrow},4^{\rightarrow},2^{\leftarrow},1^{\leftarrow}]$	4
$[1^{\leftarrow},4^{\leftarrow},2^{\leftarrow},3^{\leftarrow}]$	4	$[3^{\rightarrow},2^{\leftarrow},4^{\rightarrow},1^{\leftarrow}]$	4
$[4^{\leftarrow},1^{\leftarrow},2^{\leftarrow},3^{\leftarrow}]$	3	$[3^{\rightarrow},2^{\leftarrow},1^{\leftarrow},4^{\rightarrow}]$	3
$[4^{\rightarrow},1^{\leftarrow},3^{\leftarrow},2^{\leftarrow}]$	4	$[2^{\leftarrow},3^{\rightarrow},1^{\leftarrow},4^{\leftarrow}]$	4
$[1^{\leftarrow},4^{\rightarrow},3^{\leftarrow},2^{\leftarrow}]$	4	$[2^{\leftarrow},3^{\rightarrow},4^{\leftarrow},1^{\leftarrow}]$	4
$[1^{\leftarrow},3^{\leftarrow},4^{\rightarrow},2^{\leftarrow}]$	4	$[2^{\leftarrow},4^{\leftarrow},3^{\rightarrow},1^{\leftarrow}]$	4
$[1^{\leftarrow},3^{\leftarrow},2^{\leftarrow},4^{\rightarrow}]$	3	$[4^{\leftarrow},2^{\leftarrow},3^{\rightarrow},1^{\leftarrow}]$	3
$[3^{\leftarrow},1^{\leftarrow},2^{\leftarrow},4^{\leftarrow}]$	4	$[4^{\rightarrow},2^{\leftarrow},1^{\leftarrow},3^{\rightarrow}]$	4
$[3^{\leftarrow},1^{\leftarrow},4^{\leftarrow},2^{\leftarrow}]$	4	$[2^{\leftarrow},4^{\rightarrow},1^{\leftarrow},3^{\rightarrow}]$	4
$[3^{\leftarrow},4^{\leftarrow},1^{\leftarrow},2^{\leftarrow}]$	4	$[2^{\leftarrow},1^{\leftarrow},4^{\rightarrow},3^{\rightarrow}]$	4
$[4^{\leftarrow},3^{\leftarrow},1^{\leftarrow},2^{\leftarrow}]$	2	$[2^{\leftarrow},1^{\leftarrow},3^{\rightarrow},4^{\rightarrow}]$	

Die rekursive Natur des Algorithmus wird deutlich, wenn man den Weg des größten Elements verfolgt. Zwei benachbarte Elemente unterscheiden sich nur durch Rechtsmultiplikation mit einer Elementartransposition. Im Beispiel der S_3 erhalten wir den folgenden Weg:



Trottercode

Der Nachteil dieser Methode ist es, daß die Richtung mit abgespeichert werden muß. Um dies zu Umgehen betrachten wir nochmals die 2. Variante der rank Funktion. Im Beispiel der S_4 werden die Lehmercodes in folgender Reihenfolge erzeugt:

0000	2100	0010	2110
1000	3100	1010	3110
2000	0200	2010	0210
3000	1200	3010	1210
0100	2200	0110	2210
1100	3200	1110	3210

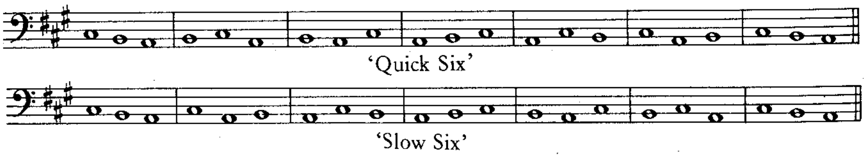
Diese Vektoren (c_3,c_2,c_1,c_0) kann man auch anders interpretieren, und so direkt die jeweilige Permutation aud dem Trotter Algorithmus rekonstruieren. Dabei wird die rekursive Struktur ausgenutzt. Im Falle der S_4 geht man z.B. wie folgt vor: c_1 (0 oder 1) codiert ob die Ziffer 2 nach der 1 kommt (0) oder nicht (1). D.h c_1 kodiert die Anzahl der kleineren rechts der 2. c_2 (0,1,2) gibt die Anzahl der Ziffern rechts der 3 (im Falle von c_1,c_0=00) oder Anzahl der kleineren Ziffern links falls c_1,c_0=10). Die allgemeine Regel wird klar bei c_3, dies gibt die Anzahl der kleineren rechts der 4, falls die Summe $c_2 + c_1 + c_0$ gerade ist, und links der 4 falls sie ungerade ist. Als Beispiel konstruieren wir die Permuation zum Vektor 233110

Vektor	Permutation	Restvektor
233110		
2331--	2,1	----10
233---	2,3,1	---110
23----	4,2,3,1	--3110
2-----	4,2,3,5,1	-33110
	4,2,3,6,5,1	

Diese Kodierung ist ebenso eine Bijektion zwischen den Permutationen und den Vektoren, und wird als *Trottercode* bezeichnet. Mit dieser Kodierung und der fallenden Faktoriellen für das ranking haben wir ein ranking, mit dem wir die Permutationen im Trotter Algorithmus durchlaufen können.

Kampanologie

Ein ähnliches Problem wie beim Trottercode stellt sich bei der Kunst des Wechselläutens[1] (Kampanologie). Dieses Hobby ist in England verbreitet und dabei geht es um das Problem n Glocken so zu läuten, daß jede Permutation einmal dabei vorkommt, im Falle von drei Glocken gibt es diese beiden verwendeten Möglichkeiten



Dabei ist es ebenso wie beim Trotter Algorithmus wichtig, daß sich benachbarte Permutationen nur geringfügig unterscheiden, damit die Harmonie im Klang erhalten bleibt. Außerdem muß die Regel von einer Permutation zur nächsten einfach sein, damit die Menschen, die Läuten nicht durcheinander kommen. Dafür haben sich Transpositionen bewährt. In dem Beispiel der S_3 war der Slow Six die Reihenfolge des Trotter Algorithmus. Im Allgemeinen werden mehrere Transpositionen verwendet um zur nächsten Permutation zu kommen, wie nachfolgende Tabelle zeigt.

1. c.f. Budden: The Fascination of Groups, Chap. 24 Groups and campanology, von dort die Bilder

Groups and campanology 465

Table 24.10 (Stedman Doubles)

1	1 2 3 4 5	31	5 4 1 3 2	61	1 2 4 3 5	91	5 3 1 4 2
2	2 1 3 5 4	32	5 1 4 2 3	62	2 1 4 5 3	92	5 1 3 2 4
3	2 3 1 4 5	33	1 5 4 3 2	63	2 4 1 3 5	93	1 5 3 4 2
4	3 2 1 5 4	34	1 4 5 2 3	64	4 2 1 5 3	94	1 3 5 2 4
5	3 1 2 4 5	35	4 1 5 3 2	65	4 1 2 3 5	95	3 1 5 4 2
6	1 3 2 5 4	36	4 5 1 2 3	66	1 4 2 5 3	96	3 5 1 2 4
7	3 1 5 2 4	37	5 4 2 1 3	67	4 1 5 2 3	97	5 3 2 1 4
8	3 5 1 4 2	38	4 5 2 3 1	68	4 5 1 3 2	98	3 5 2 4 1
9	5 3 1 2 4	39	4 2 5 1 3	69	5 4 1 2 3	99	3 2 5 1 4
10	5 1 3 4 2	40	2 4 5 3 1	70	5 1 4 3 2	100	2 3 5 4 1
11	1 5 3 2 4	41	2 5 4 1 3	71	1 5 4 2 3	101	2 5 3 1 4
12	1 3 5 4 2	42	5 2 4 3 1	72	1 4 5 3 2	102	5 2 3 4 1
13	3 1 4 5 2	43	2 5 3 4 1	73	4 1 3 5 2	103	2 5 4 3 1
14	1 3 4 2 5	44	2 3 5 1 4	74	1 4 3 2 5	104	2 4 5 1 3
15	1 4 3 5 2	45	3 2 5 4 1	75	1 3 4 5 2	105	4 2 5 3 1
16	4 1 3 2 5	46	3 5 2 1 4	76	3 1 4 2 5	106	4 5 2 1 3
17	4 3 1 5 2	47	5 3 2 4 1	77	3 4 1 5 2	107	5 4 2 3 1
18	3 4 1 2 5	48	5 2 3 1 4	78	4 3 1 2 5	108	5 2 4 1 3
19	4 3 2 1 5	49	2 5 1 3 4	79	3 4 2 1 5	109	2 5 1 4 3
20	4 2 3 5 1	50	5 2 1 4 3	80	3 2 4 5 1	110	5 2 1 3 4
21	2 4 3 1 5	51	5 1 2 3 4	81	2 3 4 1 5	111	5 1 2 4 3
22	2 3 4 5 1	52	1 5 2 4 3	82	2 4 3 5 1	112	1 5 2 3 4
23	3 2 4 1 5	53	1 2 5 3 4	83	4 2 3 1 5	113	1 2 5 4 3
24	3 4 2 5 1	54	2 1 5 4 3	84	4 3 2 5 1	114	2 1 5 3 4
25	4 3 5 2 1	55	1 2 4 5 3	85	3 4 5 2 1	115	1 2 3 5 4
26	3 4 5 1 2	56	1 4 2 3 5	86	4 3 5 1 2	116	1 3 2 4 5
27	3 5 4 2 1	57	4 1 2 5 3	87	4 5 3 2 1	117	3 1 2 5 4
28	5 3 4 1 2	58	4 2 1 3 5	88	5 4 3 1 2	118	3 2 1 4 5
29	5 4 3 2 1	59	2 4 1 5 3	89	5 3 4 2 1	119	2 3 1 5 4
30	4 5 3 1 2	60	2 1 4 3 5	90	3 5 4 1 2	120	2 1 3 4 5
e	5 4 1 3 2	s_1	1 2 4 3 5	e	5 3 1 4 2	s_1	1 2 3 4 5

random permutation

Ein weiterer wichtiger Bestandteil aller Algorithmen für kombinatorische Objekte ist die Erzeugung zufälliger Objekte. Dies ist immer dann wichtig wenn es schon so viele Objekte einer Klasse gibt, sodaß ein Durchlaufen nicht mehr möglich ist. (z.B. Permutationen vom Grad 100). Eine allgemeine Methode ist, eine zufällige Zahl zwischen 0 und Anzahl der Objekte zu erzeugen und dann mittels unrank das zugehörige Objekt zu erzeugen. Im Fall der Permutationen geht dies einfacher, man erzeugt einen zufälligen Lehmercode, und dann mittels der inversen

Abbildung die zugehörige Permutatation. Voraussetzung ist hierfür, daß man eine Routine hat, die eine Zufallszahl liefert. (z.B. rand() unter UNIX).

Zusammenfassung

Darstellung/Operation	mult	invers	next	random
Listenschreibweise	x	x	x	x
Zykelschreibweise	x	x		
Lehmercode			x	x
Trottercode			x	x
rank			x	x

Partitionen I

Partitionen sind endliche ansteigende Folgen von natürlichen Zahlen. Sei $\lambda = (0 < \lambda_1 \leq \lambda_2 \leq \ldots \lambda_k)$ eine Partition, die einzelnen λ_i, heißen *Teile* der Partition, k ist die *Länge* der Partition = l(λ), und die Summe $\Sigma\lambda_i$ heißt *Gewicht* und wird auch mit $|\lambda|$ bezeichnet. Bei der Definition von Partitionen wird von manchen Autoren auch die Anordnung als fallende Folge verwendet, und falls 0 in der Folge sind so wird dies auch manchmal erlaubt aber bei der Länge nicht mitgezählt. Diese Notation für Partitionen wird als *Listenschreibweise* bezeichnet. Es gibt auch noch eine *verkürzte Listenschreibweise*, wobei k gleiche Teile m mit m^k bezeichnet werden. Die Menge der Partitionen zu gegebenen Gewicht wird mit P_n bezeichnet, die Anzahl der Partitionen von n mit p(n).

Beispiel:

Partitionen vom Gewicht 7

7	124	11113
16	1114	1222
25	133	11122
115	223	111112
34	1123	1111111

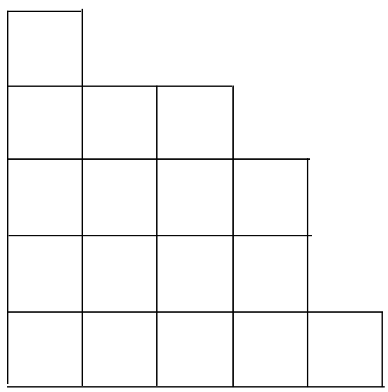
Partitionen in Symmetrica

source code	Eingabe	Ausgabe

`/* SYMMETRICA pa1.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(PARTITION,a);`
`println(a);`
`ENDE`
	
	`5`
	`1 1 1 1 3`
		`....`
		`11113`
		`....`

Ferrers Diagramm

Eine Partition $\lambda = (0 < \lambda_1 \leq \lambda_2 \leq \ldots \lambda_k)$ kann man wie folgt visualisieren. In der untersten Zeile werden λ_k 'Kästchen' gemalt, darüber λ_{k-1} , und so fort, die Kästchen werden linksbündig gezeichnet. Dieses Diagramm wird als *Ferrers Diagramm* der Partition λ bezeichnet. Die Partition 13445 hat folgendes Ferrers Diagramm:



Anzahl

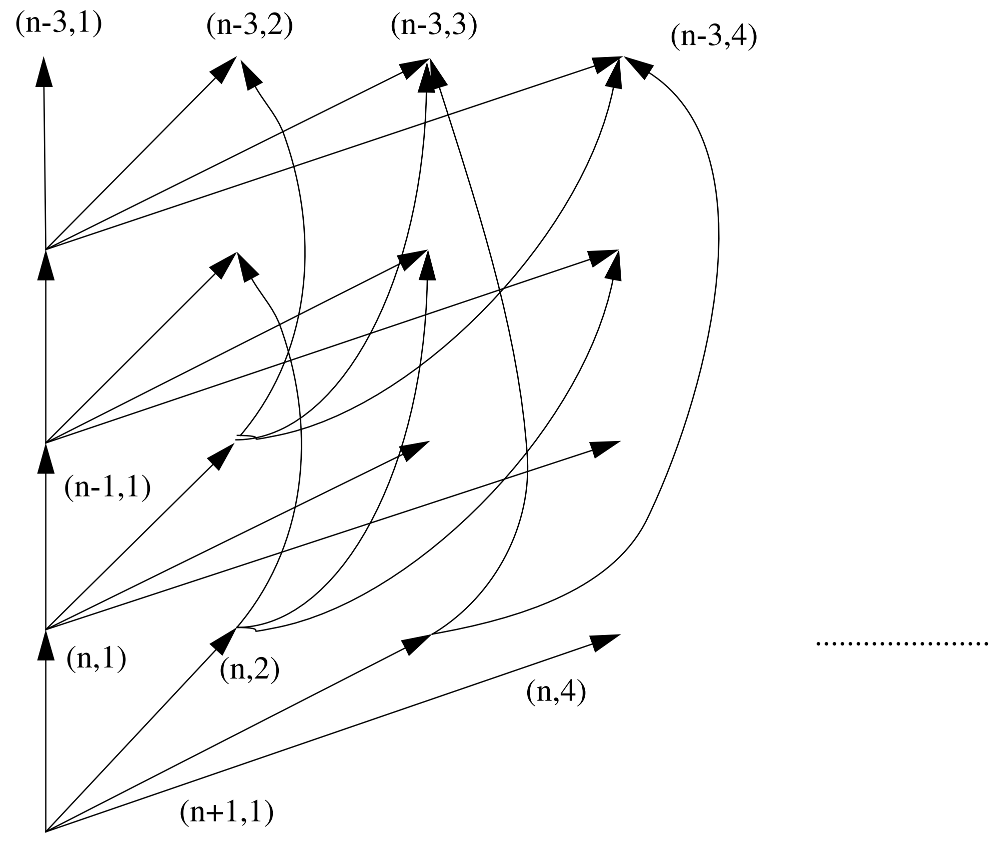
Die erste Frage bei kombinatorischen Objekten ist die Anzahl zu gegebenen Parameter. Im Falle der Permutationen war dies trivial, im Falle der Partitionen nicht mehr. Um die Anzahl $|P_n|$ zu berechnen führt man die Anzahl (m,n) ein, und bezeichnet damit die Anzahl der Partitionen von n, deren kleinster Teil = m ist. Dann kann ist $|P_n|$ = (n+1,1) = (n,1) + (n,2) + ... + (n,n). Allgemein gilt die Rekursionsformel

$$(n,k) = (n-k,k) + (n-k,k+1) + ... + (n-k,n-k)$$

Die Anzahl der Partitionen steigt sehr schnell wie folgende Tabelle zeigt:

n	Anzahl	n	Anzahl
10	42	100	190.569292
20	627	300	9253.082936.723602
30	5604	500	2300.165032.574323.995027
50	204226	1000	24.061467.864032.622473.692149.727991

Die obige Methode der Anzahlberechnung läßt sich wie folgt visualisieren:



Ordnet man die Zahlen (n,m) in einer Matrix an und notiert die Anzahl der Aufruf innerhalb der Rekursion, so erhält man folgende Matrix (Anordnung wie in obiger Visualisierung)

Anzahl der Partitonen von k mit maximal m Teilen

Anzahl der Partitionen von k mit Teilen ≤ m

n-k|m	1	2	3	4	5	6	7	8	9
n-9	1	5	12	18	23	26	28	29	30
n-8	1	5	10	15	18	20	21	22	22
n-7	1	4	8	11	13	14	15	15	15
n-6	1	4	7	9	10	11	11	11	11
n-5	1	3	5	6	7	7	7	7	7
n-4	1	3	4	5	5	5	5	5	5
n-3	1	2	3	3	3	3	3	3	3
n-2	1	2	2	2	2	2	2	2	2
n-1	1	1	1	1	1	1	1	1	1
n	1	1	1	1	1	1	1	1	1
n+1	1	0	0	0					

Daher sollte man bei der Implementierung die Ergebnisse zwischenspeichern. Der Effekt wird umso größer je größer der gespeichert Wert ist. Eine genaue Betrachtung der Rekursion ergibt, daß die Einträge (Anzahl der Aufrufe) an der Stelle (n-k,m) die Anzahl der Partitionen von k mit maximalen Teil ≤ m widerspiegelt. Betrachtet man z.B. den Eintrag an der Stelle (n-8,6), beim Aufruf dieser wurde durch die Rekursion bereits eine Partition der 8 mit maximalen Teil ≤ 6 definiert und berechnet werden soll der Rest, d.h. eine Partition von n-8 mit dem größten Teil ≤ 6. Man beachte, daß diese Tafel verschieden ist von der Tafel der (n,m) = Partitionen von n mit kleinsten Teil = m

Anzahl der Partitionen von n mit kleinsten Teil = m :

n|m	1	2	3	4	5	6	7	8	9	10
1	1	0	0	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	2	0	1	0	0	0	0	0	0	0
4	3	1	0	1	0	0	0	0	0	0
5	5	1	0	0	1	0	0	0	0	0
6	7	2	1	0	0	1	0	0	0	0
7	11	2	1	0	0	0	1	0	0	0
8	15	4	1	1	0	0	0	1	0	0
9	22	4	2	1	0	0	0	0	1	0
10	30	7	2	1	1	0	0	0	0	1

Anzahl der Partitionen in Symmetrica

In Symmetrica ist obiger Algorithmus zur Anzahlberechnung implemetiert. Folgendes Programm berechnet die gewünschte Anzahl:

source code	Eingabe	Ausgabe

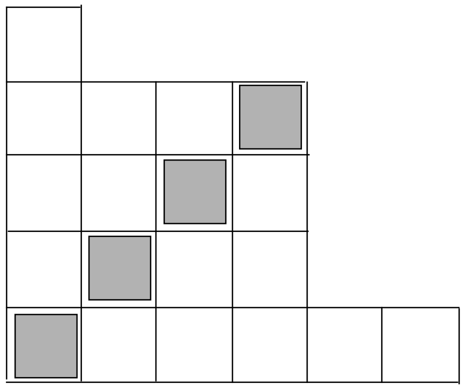
`/* SYMMETRICA pa2.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(INTEGER,a);`
`numberofpart(a,b);`
`println(b);`
`ENDE`
	`1500`	`....`
		`1329.461690.763193.8888`
		`25.263136.701886.891117`
		`....`

Verschiedene Datenstrukturen für Partitionen

Man kann Partitionen auch in der *Exponentenschreibweise* kodieren, dies ist ein Vektor $(v_1,v_2,..)$ mit Zahlen 0, wobei nur endlich viele > 0 sind. Der Eintrag an der Stelle i ist die Anzahl der Teile der Größe i in der Partition. Dies sind die Partitionen von 7 in Exponentenschreibweise(ohne 0‘en am Ende):

0000001	1101	401
100001	3001	13
01001	102	32
20001	021	51
0011	211	7

Die zweite Variante ist die *Frobenius Notation*. Hierzu betrachtet man das Ferrers Diagramm und markiert die Einträge auf der Hauptdiagonalen, dies seien k Stück. Dann bildet man zwei Vektoren der Länge k. Der erste kodiert die Anzahl der Eintrage rechts der i-ten Box, und im zweiten Vektor die Anzahl Einträge oberhalb



Die Partition 14446 wird in Frobenius Notation als (5210|4210) geschrieben.

Konjugation

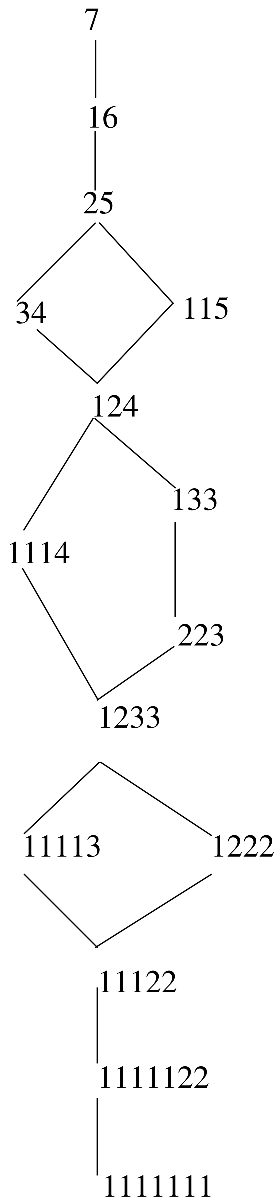
Eine wichtige Operation mit Partitionen ist der Übergang zur *konjugierten* Partition, dies entspricht das Spiegeln des Ferrers Diagramm an der Hauptdiagonalen. Die konjugierte Partition zur Partition λ wird mit λ' bezeichnet. Am leichtesten ist diese Operation in der Frobenius Notation. Die Konjugation zeigt auch, daß z.B. die Anzahl der Partitionen vom Gewicht n mit max m Teilen gleich ist der Anzahl der Partitionen n deren größter Eintrag maximal m ist.

Ordnungen

Es gibt natürlich die *lexikographische* Ordnung und die *colexikographische* Ordnung. Aber auch aus Partitionen kann man wieder interessante Posets bilden. Diesmal definiert man die $\leq$ Relation und nicht die Eigenschaft Nachbar zu sein.

$$\lambda \leq \mu :\Leftrightarrow \lambda_{l(\lambda)}+..+\lambda_{l(\lambda)-i} \leq \mu_{l(\mu)} + ... \mu_{l(\mu)-i} \text{ für alle } i \geq 0$$

Dies ist die *Dominanzordnung*, oder auch *natürliche Ordnung*, sie ergibt im Falle 7 folgendes interessantes Hasse Diagramm:



Dieses Poset wird $\mathcal{NP}_n$ (natural partition poset) bezeichnet. Dieses Poset ist ein Verband.

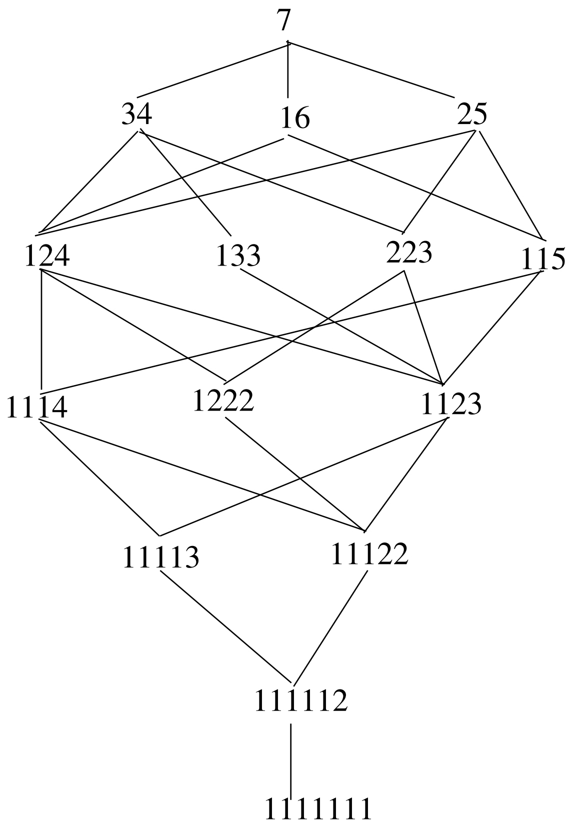
Übungsaufgabe:
Beschreibe Supremum und Infimum für zwei beliebige Partitionen aus $\mathcal{NP}_n$

Dieser Verband ist ordungs symmetrisch, besitzt keine rank Funktion, und besitzt ein Maximum und ein Minimum.

Eine weitere Ordnung erhält man indem man indem man definiert:

$$\lambda \leq \mu :\Leftrightarrow \lambda \text{ geht aus } \mu \text{ durch Aufspalten von Teilen hervor}$$

Das zugehörige Poset sieht so aus:



Dieses Poset $\mathcal{CP}_n$ (containment partition poset) ist ein Verband, hat eine rank funktion, ist nicht symmetrisch. Als rank Funktion dient die Anzahl der Teile einer Partition. In diesem Fall ist es auch einfach die Eigenschaft direkter Nachbar zu sein zu beschreiben.

Die Anzahl der Elemente in einer Schicht von $\mathcal{CP}_n$

Die Anzahl der Schichten, ist natürlich n = maximale Anzahl von Teilen. Die Anzahl der Elemente ist in der Schicht mit rank k ist die Anzahl der Partitionen von n mit genau k Teilen, dies ist unter Konjugation die Anzahl der Partition von n, deren größter Teil = k ist. Zur Berechnung dieser Zahl (n,k) kann man folgende Rekursion verwenden:

$$(n,k) = (n-k,k) + (n-k, k-1) + ... + (n-k,1)$$

Man hätte auch diese Rekursion zur Berechnung der Anzahl der Partitionen verwenden können. Die Gesamtzahl erhält man durch Summenbildung über alle k, bzw. durch Berechnung von (2n,n). Man erhält folgende Tabelle

Anzahl der Partitionen von n, deren größter Teil = k

Anzahl der Partitionen von n mit k Teilen

n|k	1	2	3	4	5	6	7	8	9	10
1	1	0	0	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0
3	1	1	1	0	0	0	0	0	0	0
4	1	2	1	1	0	0	0	0	0	0
5	1	2	2	1	1	0	0	0	0	0
6	1	3	3	2	1	1	0	0	0	0
7	1	3	4	3	2	1	1	0	0	0
8	1	4	5	5	3	2	1	1	0	0
9	1	4	7	6	5	3	2	1	1	0
10	1	5	8	9	7	5	3	2	1	1
11	1	5	10	11	10	7	5	3	2	1
12	1	6	12	15	13	11	7	5	3	2
13	1	6	14	18	18	14	11	7	5	3

Eine ähnliche Tabelle wurde bereits erstellt bei der Berechnung der Anzahl der Aufrufe bei der Berechnung der Anzahl der Partitionen von n, mit kleinsten Teil = k.

n obiger Tabelle wurde das Maximum schattiert. In Hinblick auf die Unimodalität ist die Frage in welcher Schicht die rank Funktion ihr Maximum hat.

Betrachtet man wieder die Tabelle mit den Anzahlen der Aufrufe bei der Berechnung von (n,k), so stellt man fest, daß die Anzahl der Aufrufe der Unterroutine (n-m,k-h) die Anzahl der Partitionen von n-m mit größten Teil = k und kleinsten Teil ≥ k-h ist. An der Stelle (n-m,k-h) wird eine Partition berechnet, deren größter Teil = k-h ist, d.h. die Anzahl der Aufrufe entspricht den Partitionen, der kleinster Teil ≥ k-h ist, sonst könnte man nicht mit Partitionen weitermachen,deren größter Teil = k-h ist. Aus der Startbedingung ergibt sich, daß die Anzahl der Aufrufe Partitionen entsprechen, deren maximaler Teil =k ist und vom Gewicht m dies entspricht Partitionen vom Gewicht n-m und deren maximaler Teil ≤ k ist. Diese Zahl ist gleich der Anzahl der Partitionen von n-m-k, deren größter Teil ≤ k ist und kleinsten Teil ≥ k-h ist.

Zur Veranschaulichung ein Beispiel:
Anzahl Partitionen vom Gewicht 30 mit maximalen Teil 7 = 618

n|k	1	2	3	4	5	6	7
1	522	0	0	0	0	0	0
2	436	72	0	0	0	0	0
3	364	64	17	0	0	0	0
4	300	52	15	5	0	0	0
5	248	47	14	5	2	0	0
6	201	37	11	4	2	0	0
7	164	33	10	4	1	0	0
8	131	26	9	3	1	0	0
9	105	23	7	3	1	1	1
10	82	17	6	2	1	1	0
11	65	16	6	3	2	1	0
12	49	11	4	2	1	0	0
13	38	10	4	2	1	0	0
14	28	7	3	1	0	0	0
15	21	6	2	1	0	0	0
16	15	4	2	1	1	1	1
17	11	4	2	1	1	1	0
18	7	2	1	1	1	0	0
19	5	2	1	1	0	0	0
20	3	1	1	0	0	0	0
21	2	1	0	0	0	0	0
22	1	0	0	0	0	0	0
23	1	1	1	1	1	1	1
24	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0
30	0	0	0	0	0	0	1

Die Tabelle zeigt die Anzahl der Aufrufe bei obiger Berechnung: die 3 an der Stelle (14,3) = (30-16, 3) bedeutet, daß es 3 Partitionen von 9 (=16-k) gibt, dessen kleinster Teil größer gleich 3 ist und dessen größter Teil kleiner gleich 7 (=k) ist, dies sind: 45, 36, 333. Oder, daß es 3 Partitionen von 16 gibt, dessen größter Teil gleich 7 (=k) ist, und dessen kleinster Teil größer gleich 3 , dies sind dann: 457, 367, 3337.

$\mathcal{CP}_n$ ist rank unimodal

Durchlaufen der Partitionen

Zum Durchlaufen der Partitionen wählt man die colexikographische Ordnung, man beginnt mit der Partition n. Um die nächste zu finden beginnt man mit dem kleinsten Teil und sucht das erste Teil k > 1 . Dieses Teil zusammen mit den links von ihm stehenden 1‘en bildet eine Partition von m, k wird nun zu k-1, und links davon kommt die colexikographisch erste Partition, deren größter Teil ≤ k-1 ist. Diese Methode ist die übliche Methode um kombinatorische Objekte in lexikographischer Ordnung zu durchlaufen. Eine Variation ist z.B. auch das Durchlaufen des Lehmercodes der Permutationen von gegebener Inversionen Zahl. Diese Methode läßt sich am leichtesten in der Exponentenschreibweise implementieren, da in diesem Fall die Länge des Vektors nicht wächst. Im Prinzip funktioniert er auch in der Listenschreibweise.

Durchlaufen der Partitionen in Symmetrica

In Symmetrica ist ist obiger Algorithmus zum Durchlaufen der Partitionen implemetiert. Folgendes Programm zeigt dies:

source code	Eingabe	Ausgabe

`/* SYMMETRICA pa3.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(INTEGER,a);`
`first_partition(a,b);`
`do {`
`println(b);`
`} while(next(b,b));`
`ENDE`
	
	7
		...
		7
		16
		25
		115
		34
		124
		1114
		133
		223
		1123
		11113
		1222
		11122
		111112
		1111111
		....

erzeugende Funktion

Eine prinzipielle Frage bei kombinatorischen Objekten, ist nach der erzeugenden Funktion, dies ist ein formale Potenzreihe

$$\sum c_i x^i$$

wobei der Koeffizient c_i die Anzahl der kombinatorischen Objekte zum Parameter i ist. Im Fall der Partitionen macht man sich klar, daß die erzeugende Funktion

$$(1+x+x^2+x^3+\ldots)\cdot(1+x^2+x^4+x^6+\ldots)\cdot(1+x^3+x^6+x^9+\ldots)\cdot\ldots$$

ist, jede Möglichkeit ein Produkt x^n zu bilden entspricht gerade eben einer Partition von n. Üblicherweise wird diese erzeugende Funktion geschlossen geschrieben:

$$\prod\frac{1}{(1-x^i)} = \sum p(i)x^i$$

Will man mittels dieser erzeugenden Funktion die Anzahl der Partitionen von n berechnen, so muß man die Glieder mit maximalen Grad n berücksichtigen.

erzeugende Funktion in Symmetrica

source code	Eingabe	Ausgabe

`/* SYMMETRICA pa4.c */`
`#include "def.h"`
`#include "macro.h"`
`ANFANG`
`scan(INTEGER,a);`
`copy(cons_eins,b);`
`for (i=1;i<=S_I_I(a);i++)`
`        {`
`        co_prod(i,S_I_I(a),c);`
`        mult_apply(c,b);`
`        }`
`println(b);`
`ENDE`
 	
`co_prod(int i, int n, OP c)`
`/* c = 1 + x^i + x^2i + .. */`
`/* bis max grad n */`
`{`
`        OP d;`
`        INT j;`
`        init(MONOPOLY,c);`
`        for (j=0;j*i <= n; j++)`
`        {`
`        d = callocobject();`
`        b_skn_mp(`
`          callocobject(),`
`          callocobject(),NULL,d);`
`        m_i_i(1,S_PO_K(d));`
`        m_i_i(j*i,S_PO_S(d));`
`        insert(d,c,NULL,NULL);`
`        }`
`}`
	7	...
		
		1 0 1 1 2 2 3 3 5 4 7 5
		11 6 <u>15 7</u> 18 8 24 9 30 10
		37 11 44 12 52 13 57 14 64
		15 71 16 77 17 81 18 84 19
		84 20 84 21 84 22 81 23 77
		24 71 25 64 26 57 27 52 28
		44 29 37 30 30 31 24 32 18
		33 15 34 11 35 7 36 5 37
		3 38 2 39 1 40 1 41
		....

Die Unterroutine co_prod(i,n,c) berechnet die Summe $c = 1 + x^i + x^{2*i} + ... x^{k*i}$ solange $k*i \leq n$ ist. Das Ergebnis ist das Produkt der in co_prod berechneten Summen, zu lesen als Paare Koeffizient, Exponent. Das Paar 15 7 (im Ergebnis unterstrichen) bedeutet dann, daß es 15 Partitionen von 7 gibt. Betrachtet man noch mal die erzeugende Funktion, so sieht man, daß das Gesamtergebnis die Anzahl der Partitionen abzählt die aus max 7 1'en, 3 2'en, 2 3'en, einer 4, 5, 6, 7 gebildet werden kann. Es gibt also 5 Möglichkeiten daraus eine Partition von 37 zu bilden. Dies sind auch die 5 Partition von 4, die dabei verwendeten Teile werden einfach abgezogen, so sieht man auch, daß die Zahlenfolge der Koeffizienten symmetrisch ist.

erzeugende Funktionen für Partitionen mit speziellen Eigenschaften

Das Symmetrica Beispiel zeigte wie eine Einschränkung der erzeugenden Funktion zu interpretieren ist. Will man die Anzahl der Partitionen mit maximal k Teilen (= größte Teil = k) so schränkt man die erzeugende Funktion so ein:

$$\frac{1}{(1-x)\cdot(1-x^2)\cdot\ldots\cdot(1-x^k)} = \sum p_{\leq k}(i)x^i$$

Will man die Anzahl der Partitionen, deren größter Teil = k ist, so wird der Faktor x^k hinzu multipliziert:

$$\frac{x^k}{(1-x)\cdot(1-x^2)\cdot\ldots\cdot(1-x^k)} = \sum p_{=k}(i)x^i$$

Das letzte Beispiel war die Anzahl der Partitionen mit kleinsten Teil h und maximalen Teil≤ k, die erzeugende Funktion ist:

$$\frac{1}{(1-x^h)\cdot(1-x^{h+1})\cdot\ldots\cdot(1-x^k)} = \sum p_{h\leq k}(i)x^i$$

Permutationen II

Permutationen als Gruppenelemente

Wie bereits gesehen bilden die Permutationen von festem Grad eine Gruppe, in diesem Abschnitt behandeln wir gruppentheoretische Fragen.

Konjugiertenklassen

D.h. gesucht wird eine Charakterisierung der Elemente aus *C(*σ*)*$=\{\pi \in S_n |$ mit $\pi = \rho\sigma\rho^{-1}$ wobei $\rho \in S_n\}$, der *Konjugiertenklasse* von σ. Dazu betrachtet man ein Element aus *C(*σ*)* in der langen Listenschreibweise.

$$\begin{bmatrix} \sigma(i) \\ \rho(\sigma(i)) \end{bmatrix} \cdot \begin{bmatrix} i \\ \sigma(i) \end{bmatrix} \cdot \begin{bmatrix} \rho(i) \\ i \end{bmatrix}$$

D.h. Aus der Permutation

$$\begin{bmatrix} i \\ \sigma(i) \end{bmatrix}$$

entsteht die Permutation

$$\begin{bmatrix} \rho(i) \\ \rho(\sigma(i)) \end{bmatrix}$$

um das Element unter Konjugation mit ρ zu bekommen muß man in der Listenschreibweise in beiden Zeilen die Einträge durch ihre Bilder unter ρ ersetzen. Konjugation bedeutet nichts anderes als das Umnumerieren der beteiligten Zahlen 1...n.
Beispiel:

$$\sigma = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{bmatrix}, \rho = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{bmatrix}, \rho\sigma\rho^{-1} = \begin{bmatrix} 3 & 2 & 4 & 1 \\ 2 & 4 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 1 \end{bmatrix}$$

Betrachte man die Konjugation in Zykelschreibweise, so wird der Zykelteil (...., i , σ(i) , ...) ersetzt durch (....,ρ(i), $\rho(\sigma(i))$, ...), d.h. die Längen der einzelnen disjunkten Zykeln bleiben erhalten. Umgekehrt kann man zeigen, daß zwei Permutationen mit gleichen Zykellängen konjugiert sind. Sei dazu als Beispiel in Zykelschreibweise:
σ = (1,3,5)(2,6)(4,7) und π = (3,4,5)(1,2)(6,7) so sind diese mittels ρ = [3,1,4,6,5,2,7] konju-

giert. Damit wurde gezeigt, daß die Zykellängen die Konjugiertenklasse beschreiben. Dazu definiert man den *Zykeltyp* einer Permutation π aus S_n, dies ist ein Vektor $a(\pi) = (a_1(\pi),...,a_n(\pi))$ mit a_i:= Anzahl der Zykel der Länge i in π. Mit $c(\pi)$ bezeichnet man die *Anzahl der Zykel* von π. Eine Variante ist die Verwendung der Zykelpartition einer Permutation, dies ist eine ansteigende Folge von $c(\pi)$ Zahlen, wobei die Zahl x genau $a_x(\pi)$ mal vorkommt, die Summe über diese Zahlen ist n, und derartige Zerlegungen von n heißen *Partitionen*.

Beispiel:

Permutation	Zykelschreibweise	Zykeltyp	Zykelpartition
[2,1,4,5,3,7,6]	(3,4,5)(1,2)(6,7)	(0,2,1,0,0,0,0)	(2,2,3)
[3,1,4,6,5,2,7]	(1,3,4,6,2)(5)(7)	(2,0,0,0,1,0,0)	(1,1,5)

Visualisierung von Permutationen

Eine im Hinblick auf weitere Anwendungen nützliche Darstellungsweise von Permutationen π, ist die sog. *Permutationsmatrix*, dies ist eine 0-1 Matrix, mit 1 genau in den Koordinaten $(i,\pi(i))$. Beispiel: zur Permutation [2,1,4,5,3,7,6] gehört die Matrix

$$\begin{bmatrix} 0&1&0&0&0&0&0\\ 1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0\\ 0&0&1&0&0&0&0\\ 0&0&0&0&0&0&1\\ 0&0&0&0&0&1&0 \end{bmatrix}$$

Das wichtige an dieser Darstellung ist die Tatsache, daß die Matrizen selber eine Gruppe bilden und diese Abbildung eine gruppenverträgliche Abbildung ist, d.h. ein Homomorphismus. D.h. man kann Multiplikation und Invertieren auf der Ebene der Matrizen und später wieder zurück zu den Permutationen wechseln.

source code	Eingabe	Ausgabe

`/* SYMMETRICA p8.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(PERMUTATION,a);`
`perm_matrix(a,b);`
`println(b);`
`ENDE`
	`7`
	`2 1 4 5 3 7 6`
		`....`
		
		`[0:1:0:0:0:0:0:]`
		`[1:0:0:0:0:0:0:]`
		`[0:0:0:1:0:0:0:]`
		`[0:0:0:0:1:0:0:]`
		`[0:0:1:0:0:0:0:]`
		`[0:0:0:0:0:0:1:]`
		`[0:0:0:0:0:1:0:]`
		`....`

Nützlich ist folgende Beobachtung, die inverse Matrix erhält man hier durch transponieren, dies bedeutet, daß die Involutionen gerade die Permutationen sind, deren Permutationsmatrix symmetrisch an der Hauptdiagonale ist.

Rothediagramm

Obige Visualisierung ist auch sehr nützlich im Zusammenhang mit den Inversionen einer Permutation. Dazu zieht man in der Permutationsmatrix ausgehend von den 1‘en nach rechts und unten Linien, die nicht gestrichenen 0‘en zählen dann die Inversionen.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Dieses Diagramm mit Linien 0‘en und 1‘en bleibt gültig, wenn man zur inversen Permutation übergeht, und so sieht man, daß die Anzahl der Inversionen invariant ist unter der Operation des Invertierens. Den Lehmercode der Permutation erhält man, wenn man in die Anzahl der Inversionen in den Zeilen zählt. Die anderen Inversionstabellen erhält man, wenn man die Inversionen in den Spalten zählt, oder nicht die übrig gebliebenen 0‘en zählt, sondern die doppelt gestrichenen 0‘en. Dieses Diagramm, mit welchem man die Inversionen zählen kann, heißt *Rothediagramm*. Diese Notation wurde bereits 1800 von H.A. Rothe[1] verwendet. Auch Rothe untersuchte Inversionen.

Inversion Poset

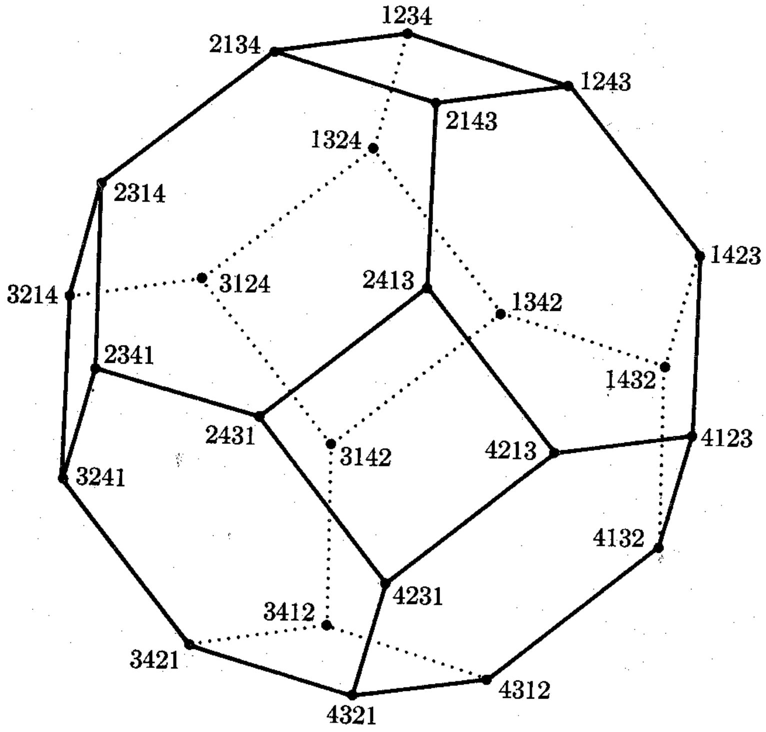
Ein *Poset* (partial ordered set) ist eine Menge P von Objekten {a,b,c,...} zusammen mit einer Ordung $\leq$ die folgende drei Eigenschaften hat:

1. $a \leq a$ für alle a aus P
2. aus $a \leq b$ und $b \leq a$ folgt $a = b$
3. aus $a \leq b$ und $b \leq c$ folgt $a \leq c$

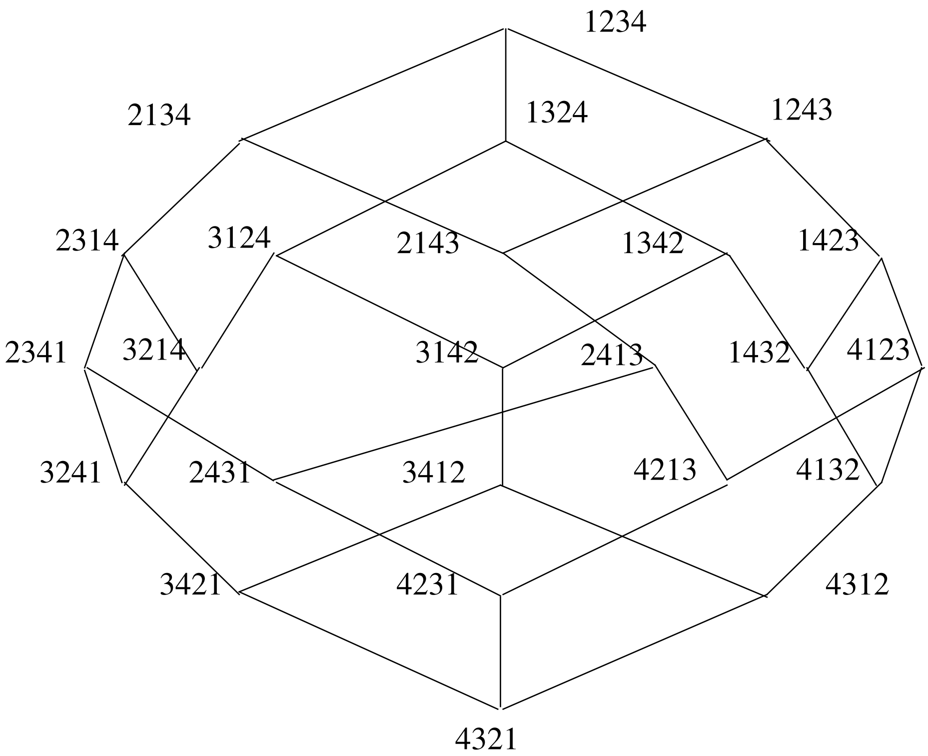
Man schreibt $a < b$ wenn gilt $a \leq b$ und a und b verschieden sind. Um ein Poset zu untersuchen genügt es wegen 3 sich um *direkte Nachbarn* in der Ordnung zu kümmern, d.h. man schreibt $a <\cdot\, b$ wenn wenn es kein c gibt mit $a<c<b$ und $a<b$ gilt. Ein Poset wird visualisiert durch einen Graphen des Knoten die Objekte a,b,c... sind und Kanten werden zwischen den direkten Nachbarn gezogen. Dies ist das sog. *Hassediagramm* des Posets. Der wichtige Schritt ist stets das definieren der Ordnung, dabei wird meist nur die Eigenschaft direkter kleinerer Nachbar zu sein definiert. Für Permutationen gibt es verschiedene Ordnungen, im ersten Beispiel sagen wir: zwei Permutationen sind benachbart($\pi <\cdot\, \tau$), wenn sie durch (Rechts-)Multiplikation mit einer Elementartransposition σ_i auseinander hervorgehen und $\pi_i < \pi_{i+1}$: Im Falle der S_n erhalten wir

1. H.A. Rothe in C.F. Hindenburg: Sammlung combinatorisch-analytischer Abhandlungen Leipzig 1800

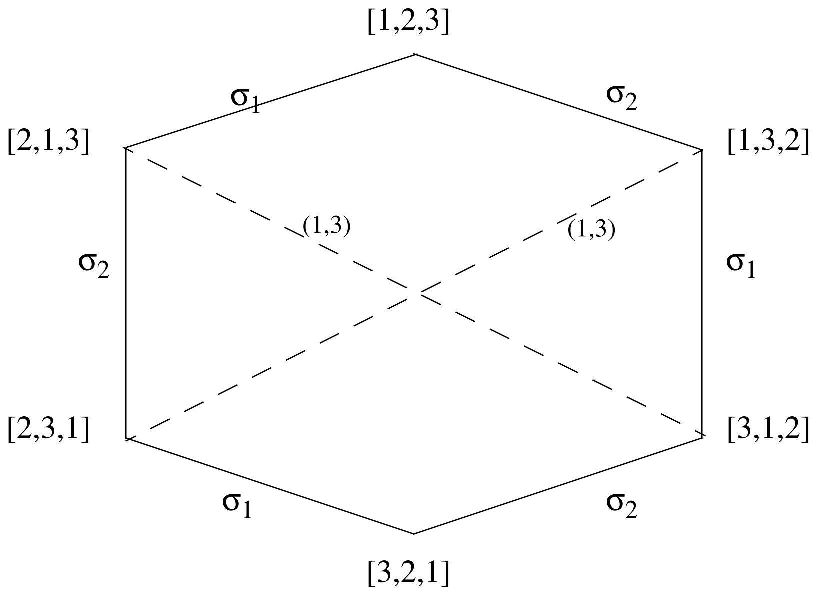
folgendes Poset (aus Knuth III) :Dies erhält den Namen *Inversion Poset* I_n. Es handelt sich um



einen Würfel, dessen Ecken so abgeschnitten wurden, daß Sechsecke entsanden sind. Bei diesem Poset gibt es ein eindeutiges größtes und kleinestes Element, genannt *Minimum* und *Maximum*. Es handelt sich um die Permutationen [1,2,...,n] und [n,n-1,...,2,1]. Eine weitere wichtige Eigenschaft eines Posets ist die Möglichkeit, dieses Poset mit einer rank Funktion zu versehen, dies ist wieder eine Abbildung r nach {1,2,...} (diesmal keine Bijektion) und für benachbarte Elemente a <· b gilt r(b) = r(a)+1. Dann kann man die Objekte a,b,c,.. in Schichten anordnen. Ein solches poset heißt dann *ranked Poset*. Gibt es ein Minimum, so ist dies in der *Schicht* mit dem niedrigsten rank, das Maximum in der Schicht mit maximalen rank. Das Inversion Poset ist ein Poset mit einer rank-Funktion, man zählt hierzu die Anzahl der Inversionen in einer Permutation, und erhält so die Ebene zum zeichnen. Das Bild oben wurde unter optischen Gesichtspunkten ausgewählt, nicht nach der rank Funktion:



An diesem Beispiel sieht man noch weitere Eigenschaften eines Posets. Dazu betrachtet man die Zahlenfolge, die man erhält, wenn man die Anzahl der Objekte innerhalb einer Schicht zählt. Im obigen Beispiel ist dies die Zahlenfolge (1,3,5,6,5,3,1). Diese Zahlenfolge ist *unimodal*, d.h. von der Form $(w_1 \leq w_2 \ldots \leq w_m \quad w_{m+1} \ldots \quad w_k)$. Ein endlicher ranked Poset, heißt dann *rank unimodal*. Ist diese Folge auch noch symmetrisch zur Mitte, so heißt das ranked Poset *rank symmetrisch*. Und erhält man durch Spiegeln eines endlichen Posets and der Mitte, das gleiche Poset wieder, so heißt das Poset *ordnungs symmetrisch*. All dies sind Eigenschaften, die vom Inversion Poset erfüllt werden. Ein Poset heißt *Verband*, wenn es zu zwei beliebigen Elementen a und b ein eindeutiges kleinstes Element $(a \vee b)$ gibt welches größer ist als a und b, das *Supremum* und ein eindeutiges größtes Element $(a \wedge b)$ gibt welches kleiner ist als a und b, das *Infimum*. Das Inversion Poset ist ein Verband. Als Beispiel betrachten wir eine Variation des Inversion Poset, zwei Permutationen seien benachbart, wenn sie durch eine beliebige Transposition auseinander hervorgehen:



Durch die gegenüber den ursprünglichen Inversion Poset neu hinzugekommenen Relationen (gestrichelt) haben z.B. die beiden Permutationen [1,3,2] und [2,1,3] kein eindeutiges Element, welches ein Supremum darstellt. Dieses Poset mit zusätzlichen Relationen heißt *starkes Inversion Poset* $\mathcal{S}I_n$, zur Unterscheidung wird das Poset nur mit Elementartranspositionen auch *schwaches Inversion Poset* genannt. Diese Ordnung (mit beliebigen Transpositionen) heißt *(starke) Bruhat Ordnung*. Die Ordnung des schwachen Inversion Poset heißt *schwache Bruhat Ordnung*.

Anzahl der Schichten, Elemente in einer Schicht

Die Anzahl der Schichten im Inversion Poset I_n ist

$$(n-1)+(n-2)+...+1=\binom{n}{2}.$$

Die nächste Frage ist wieviel Objekte sind in der Schicht k. D.h. wieviel Permutationen gibt es vom Grad n mit k Inversionen. Dazu betrachtet man den Lehmer Code, die Summe über die Einträge ergit die Anzahl der Inversionen: Um die Zahl (n,k) = Anzahl der Permutationen vom Grad n mit k Inversionen zu berechnen, verwendet man die Rekursion:

$$(n,k) = (n-1,k) + (n-1,k-1) + ... + (n-1,k-n+1)$$

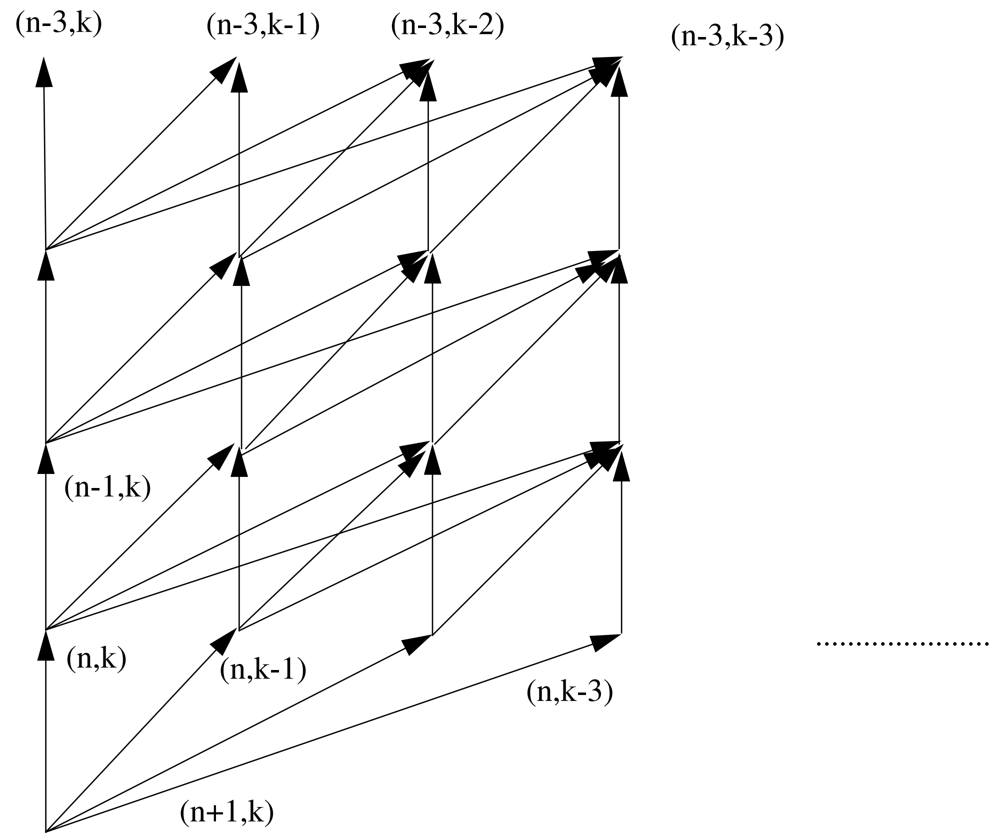
mit der Konvention (n,h) = 0 falls $h < 0$. Die Rekursion ergibt sich aus den Möglichkeiten aus einem Lehmercode der S_{n-1} einen Lehmercode der S_n zu bauen, indem man an der ersten Stelle eine Zahl zwischen 0 und n-1 einsetzt.

Anzahl der Permutationen von n mit k Inversionen

Die Anzahl steigt schnell, wie folgende Tabelle zeigt:

n|k	0	1	2	3	4	5	6	7	8	9	10
1	1	0	0	0	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0	0
3	1	2	2	1	0	0	0	0	0	0	0
4	1	3	5	6	5	3	1	0	0	0	0
5	1	4	9	15	20	22	20	15	9	4	1
6	1	5	14	29	49	71	90	101	101	90	71
7	1	6	20	49	98	169	259	359	455	531	573
8	1	7	27	76	174	343	602	961	1415	1940	2493
9	1	8	35	111	285	628	1230	2191	3606	5545	8031
10	1	9	44	155	440	1068	2298	4489	8095	13640	21670
15	1	14	104	545	2260	7889	24087	66013	165425	384320	836604
20	1	19	189	1310	7105	32110	125761	438407	1386735	4038090	10947079

Die Maxima der Anzahlen wurden mittels Schattiering markiert. Da das Inversion Poset rank symmetrisch und rank unimodal ist, sind diese Zahlenfolgen symmetrisch zur Mitte und auch unimodal. Die obige Rekursion läßt sich visualisieren:



Ordnet man die Zahlen (n,k) in einer Matrix an und notiert die Anzahl der Aufrufe innerhalb der Rekursion, so erhält man folgende Matrix (Anordnung wie in obiger Visualisierung)

n-m|k-h	k	k-1	k-2	k-3	k-4	k-5	k-6	k-7	k-8
n-8	1	9	45	165	495	1287	3003	6435	12870
n-7	1	8	36	120	330	792	1716	3432	6435
n-6	1	7	28	84	210	462	924	1716	3003
n-5	1	6	21	56	126	252	462	792	1287
n-4	1	5	15	35	70	126	210	330	495
n-3	1	4	10	20	35	56	84	120	165
n-2	1	3	6	10	15	21	28	36	45
n-1	1	2	3	4	5	6	7	8	9
n	1	1	1	1	1	1	1	1	1
n+1	1	0	0	0	0	0	0	0	0

Der Eintrag an der Stelle (m,h) ist die Anzahl der Möglichkeiten die Zahl h als Summe von m+1 positven Zahlen zu schreiben. Z.B. (2,3) = 10 = (003,012,021,030,102,111,120,201,210,300). Dies wird aus der verwendeten Rekursion klar, an der Stelle (m,h) wird versucht die Anzahl der Permutationen von S_{n-m} mit k-h Inversionen zu berechnen, daß heißt bisher wurden die die ersten m Stellen des Lehmer codes belegt, sodaß die Summe der Einträge h ist. Derartige kombinatorische Objekte heißen *uneigentliche Partitionen*, oder ungeordnete Partitionen. Der Eintrag an der Stelle (m,h) ist die Anzahl der uneigentlichen Partitionen von h in m+1 Teilen.

Übungsaufgabe:
Zeige, daß der Eintrag an der Stelle (m,h)=:C(m,h)=$\binom{m+h}{h}$

Durchlaufen der Permutationen von gegebenen Grad und gegebener Anzahl von Inversionen

Dies geht relative einfach indem man in lexikographischer Reihenfolge die Lehmer Codes der entsprechenden Permutationen erzeugt, nehmen wir als Beispiel die S_4 und die Permutation mit 3 Inversionen

Lehmercode	Permutation	Lehmercode	Permutation
3000	4123	1200	2413
2100	3214	1110	2341
2010	3142	0210	1432

Erzeugen einer zufälligen Permutation von gegebenen Grad und gegebener Anzahl von Inversionen

Hierzu kann man ein allgemeines Verfahren anwenden, welches immer zur Verfügung steht wenn man eine Rekursion zur Anzahlberechnung verwendet. Das Verfahren läßt sich allgemein so beschreiben:

EINGABE:	Grad n, Anzahl der Inversionen k
AUSGABE:	Zufällige Permutation zu diesen Parametern
ALGORITHMUS:	I: Berechne die Matrix (n,k) nach bekannter Rekursion (n,k) = (n-1,k) + (n-1,k-1) + ... II: Erzeuge eine Zufallszahl p zwischen 1 und |(n-k)| III: Bestimme größtes j sodaß p ≤ (n-1,k) + ... + (n-1,k-j) IV: der Lehmercode der gesuchten zufällige Permutation beginnt mit j. V: Wende falls n > 2 die Rekursion ab II an mit k = k-j und p = p - (n-1,k) - .... -(n-1,k-j+1) und n = n-1 ENDE

erzeugende Funktion für die Anzahl der Permutationen von n mit k Inversionen

Man überlegt sich folgende erzeugende Funktion

$$1 \cdot (1+x) \cdot (1+x+x^2) \cdot \ldots \cdot (1+x+\ldots+x^{n-1}) = \sum i_k x^k$$

Der Koeffizient i_k ist dann die Anzahl der Permutationen von n mit k Inversionen. Sie ergibt sich direkt aus der verwendeten Rekursion.

Wie kann man zwei beliebige Permutation bezüglich der beiden Bruhat Ordnungen vergleichen?

Bisher wurde nur die Eigenschaft Nachbar zu sein definiert, die Frage ist wie kann ich entscheiden, in welcher Relation zwei beliebige Permutationen stehen.

Tableaux

Ein *Tableau t* ist eine Füllung der Boxen eines Ferrers Diagramm einer Partition λ mit Buchstaben eines Alphabets A = $\{a_1 < a_2 < ..\}$. üblicherweise verwendet man dazu die Zahlen 1,2,..... Diese Buchstaben sind die Einträge. Dabei werden die Boxen mit ihren Koordinaten bezeichnet, d.h. man spricht vom Eintrag am Platz (i,j)

Beispiel:

1				
1	2	5		
2	3	3	4	
2	5	7	8	
2	4	4	6	11

Der Eintrag am Platz (1,4) ist z.B. eine 6.
Die Partition λ ist der *Umriß* des Tableau. Das *Gewicht* des Tableau (= |t|) ist das Gewicht der Partition, d.h. die Anzahl der Einträge im Tableau. Der *Inhalt* c(t) des Tableau ist der Vektor $(...,g_i,..)$ wobei g_i angibt wie oft das i-te Zeichen des Alphabets im Tableau vorkommt. Im obigen Beispiel ist der Inhalt = (2,4,2,3,2,1,1,1,0,0,1,0,...). Der Inhalt ist eine uneigentliche Partition vom Gewicht des Tableau. Ein Tableau heißt Young Tableau, wenn der Inhalt (1,1,1,1,....) ist. D.h. jeder der ersten |t| Buchstaben kommt genau einmal vor. Ein Tableau heißt *standard*, wenn die Einträge in den Zeilen schwach monoton steigen und in den Spalten streng monoton.
Dies sind alle standard Tableaux vom Umriß (1,2) in einem Alphabet {a,b,c}

$$\begin{matrix} b \\ a\,a \end{matrix}, \begin{matrix} c \\ a\,a \end{matrix}, \begin{matrix} b \\ a\,b \end{matrix}, \begin{matrix} c \\ a\,b \end{matrix}, \begin{matrix} b \\ a\,c \end{matrix}, \begin{matrix} c \\ a\,c \end{matrix}, \begin{matrix} c \\ b\,b \end{matrix}, \begin{matrix} c \\ b\,c \end{matrix}$$

Von diesen 8 standard Tableaux, sind nur zwei Standard Young Tableaux. Die Menge der Standard Young Tableaux zum Umriß λ wird mit SYT(λ) bezeichnet.

Tableaux in Symmetrica

source code	Eingabe	Ausgabe

`/* SYMMETRICA pa1.c */`
`#include "def.h"`
`#include "macro.h"`
 	
`ANFANG`
`scan(TABLEAUX,a);`
`println(a);`
`ENDE`
	
	`P`
	`3 1 2 3`
	`1 3 5`
	`2 4`
	`6`
		`....`
		`6`
		`2 4`
		`1 3 5`
		`......`

Dabei wird mit P festgelegt, daß als Umriß eine Partition verwendet wird, und diese wird dann eingegeben, nachfolgend die Einträge des Tableau.

weitere Kodierung für Standard Young Tableaux

Zu einem SYT t mit n Einträgen wird eine n elementige Folge y(t) definiert. Der i-te Eintrag gibt die Zeilennummer des Eintrags i im Tableau. Dies ist das sog. *Yamanouchi Wort* des Tableau. Zum Tableau t

12				
6	16	17		
3	13	14	15	
2	5	8	10	
1	4	7	9	11

gehört das Wort:

y(t)=12312412121533344

Die Worte die zu einem SYT gehören werden als *lattice permutation* bezeichnet. Es sind gerade die Permutationen der lex. kleinsten Folge 1....12....23.....3...., die einen Weg im Young Verband von der leeren Partition zum Umriß des SYT kodieren. Die Abbildung y ist also eine Bijektion zwischen den SYT und den lattice permutations.

nächste Standard Young Tableaux [Nijenhuis Wilf]

Die lattice permutations sind die richtige Kodierung um ein nächstes SYT zu berechnen. Zunächst definiert man die Reihenfolge mit SYT. Das erste sei das Tableau welches man durch spaltenweise Füllen erhält.

5				
4	9	13		
3	8	12	16	
2	7	11	15	
1	6	10	14	17

Das letzte sei das Tableau mit der zeilenweisen Füllung. Um zu einem SYT das nächste zu finden betrachtet man die Subtableau aus den Zahlen 1...i, und sucht das kleinste i, sodas das Subtableau nicht das letzte ist. Z.B. im Tableau

5	7	12		
3	4	8	10	
1	2	6	9	11

ist die Zahl i=6. Diese Zahl ist dann nicht in der obersten Ecke, und sie wird in die nächste Ecke nach oben geschoben. Die übrig gebliebenen Kästchen im Subtableau werden spaltenweise mit den Zahlen 1...i-1 aufgefüllt, d.h. mit dem ersten der gewünschten Reihenfolge der Tableau. Im Beispiel ist dies das Subtableau

3		
2	6	
1	4	5

die Kästchen mit den Einträgen i+1,... bleiben unverändert. Im Beispiel erhält man das SYT

3	7	12		
2	6	8	10	
1	4	5	9	11

Es ist klar, daß man so alle SYT durchläuft. (Induktion). Man beobachtet dazu daß man alle SYT zur Partition λ erhält, wenn man alle Tableaux zu Umrissen betrachte, die man aus λ erhält indem man eine Eckbox entfernt. Dies sind die kleineren Nachbarn zu λ im Young Verband.

Zur Implementierung von next ist es einfacher SYT durch lattice permutations zu kodieren. Der Algorithmus sieht dann so aus:

EINGABE:	lattice permutation $[v_1,....,v_n]$
AUSGABE:	nächste lattice permutation $[v_1,....,v_n]$
ALGORITHMUS:	I: suche das kleinste i mit $v_i < v_{i-1}$ II: falls es kein solches i gibt, hat man bereits die letzte ENDE III: ersetze v_i durch die Zeilennummer der nächst höheren Ecke im Subtableau IV: ersetze $v_1,...,v_{1-1}$ mit lex. erster lattice permutation: 123..123..12.. ENDE

Beispiel zum Durchlauf der SYT

SYT	lattice permutation	SYT	lattice permutation
3 2 5 1 4 6	123121	5 2 6 1 3 4	121132
4 2 5 1 3 6	121321	5 3 6 1 2 4	111232
4 3 5 1 2 6	112321	5 4 6 1 2 3	111232
5 2 4 1 3 6	121231	6 2 4 1 3 5	121213
5 3 4 1 2 6	112231	6 3 4 1 2 5	112213
3 2 6 1 4 5	123112	6 2 5 1 3 4	121123
4 2 6 1 3 5	121312	6 3 5 1 2 4	112123
4 3 6 1 2 5	112312	6 4 5 1 2 3	111223

Dies ist die colexikographische Ordnung auf den lattice permutations.

Zufallserzeugung von SYT

Dazu betrachte man nochmal die Rekursion zur Berechnung der Anzahl f_λ der SYT vom Umriß λ.

$$f_\lambda = \sum f_\mu$$

Dabei wird über alle Partitionen μ addiert die aus λ durch Entfernen einer Eckbox entstehen.Diese rekursive Berechnungsmethode kann wieder nach der Standardmethode wie bei Partitionen oder Permutationen mit fester Anzahl von Inversionen zur Zufallserzeugung verwendet werden.

Beispiel zur Zufallserzeugung von SYT

Gesucht ist ein zufälliges SYT vom Umriß 1357, dazu wird zuerst die Anzahl f_{1357} berechnet, es gilt

$f_{1357} = f_{357} + f_{1257} + f_{1347} + f_{1356} = 45045 + 108108 + 135135 + 128700 = 416988$

Nun wird eine Zufallszahl zwischen 0 und 416988 z.B. 300000 bestimmt, und damit ist die Position des maximalen Eintrags, der 16 festgelegt, denn das 300000-te SYT kommt aus dem 4-ten Teil, d.h. den Tableaux die durch Anfügen einer Box an ein Tableau vom Umriß 1356 entstehen. Das restliche Verfahren wird wie folgt weiter durchgeführt:

	Vorgänger Anzahl	Vorgänger Anzahl	Vorgänger Anzahl	Vorgänger Anzahl	SYT
1357					
416988 300000	357	1257	1347	1356	
	45045	108108 153153	135135 288288	128700	 16
1356					
128700 11712	356				
	15015				15 16
356					
15015 11712	256	346	355		
	5148	6435 11583	3432		15 . . . . 14 16
355					
3432 129	255				
	1320				15 . . 13 . . . . 14 16
255					
1320 129	155				
	330				15 . 12 13 . . . . 14 16

	Vorgänger Anzahl	Vorgänger Anzahl	Vorgänger Anzahl	Vorgänger Anzahl	SYT
155					
330 129	55	145			
	42	288 330			15 . 12 13 . . . . 11 . . . . . 14 16
145					
288 87	45	135			
	42	162 204			15 . 12 13 . . . 10 11 . . . . . 14 16
135					
162 45	35	125			
	28	64 109			15 . 12 13 . . 9 10 11 . . . . . 14 16
125					
64 17	25	115			
	14	15 29			15 . 12 13 . 8 9 10 11 . . . . . 14 16
115					
15 3	15				
	5				15 7 12 13 . 8 9 10 11 . . . . . 14 16
15					
5 3	5	14			

	Vorgänger Anzahl	Vorgänger Anzahl	Vorgänger Anzahl	Vorgänger Anzahl	SYT
	1	4			15 7 12 13 . 8 9 10 11 . . . . 6 14 16
14					
4 2	4	13			
	1	3			15 7 12 13 . 8 9 10 11 . . . 5 6 14 16
13					
3 1	3				
	1				15 7 12 13 4. 8 9 10 11 . . . 5 6 14 16
3					
1 1	2				
	1				15 7 12 13 3. 8 9 10 11 . . 4 5 6 14 16
2					
1 1	1				
	1				15 7 12 13 3. 8 9 10 11 . 2 4 5 6 14 16

Das 300000-te SYT vom Umriß 1357 ist also das Tableau

```
15
 7  12 13
3.  8  9  10 11
 1  2  4  5  6  14 16
```

Zweites Verfahren zur Zufallserzeugung von SYT [Nijenhuis Wilf]

Das vorgestellte Verfahren hat den üblichen Nachteil des Standardverfahrens, es muß die Anzahl des gesuchten kombinatorischen Objekts mittels einer Rekusion vorbestimmen. Es wird nun ein Verfahren vorgestellt, daß ohne diese Anzahlberechnung auskommt.

EINGABE:	Umriß λ vom Gewicht n
AUSGABE:	zufälliges SYT t vom Umriß λ
ALGORITHMUS:	I: m=n, erzeuge t mit leeren Einträgen II: füge die Zahl m in ein leeres Kästchen $\mathcal{K}$ von t III: sei h die Anzahl der leeren Kästchen rechts und oberhalb von $\mathcal{K}$. IV: ist h > 1 schiebe m in eines der leeren Kästchen rechts und oberhalb von $\mathcal{K}$ gehe wieder zum Schritt III V: ist h = 0 und m > 1, erniedrige m und gehe wieder zum Schritt II ENDE

Beispiel zum zweiten Verfahren

Es soll ein zufälliges Tableau vom Umriß 1357 erzeugt werden, man beginnt mit dem leeren Tableau

```
.
. . .
. . . . .
. . . . . . .
```

Im Schritt II wird die 16 zufällig in einem der leeren Kästchen plaziert:

```
.
. . .
16 . . . .
. . . . . . .
```

jetzt sind insgesamt 6 freie Kästchen recht und oberhalb, eines dieser wird zufällig ausgewählt, und die 16 wird dorthingeschoben(IV):

```
.
. . .
. . 16 . .
. . . . . . .
```

nun sind 3 freie Kästchen recht und oberhalb, eines dieser wird zufällig ausgewählt, und die 16 wird dorthingeschoben:

```
.
. . 16
. . . . .
. . . . . . .
```

nun ist die 16 in einer Ecke(V) und es wird mit der Zahl 15 weitergemacht u.s.w.

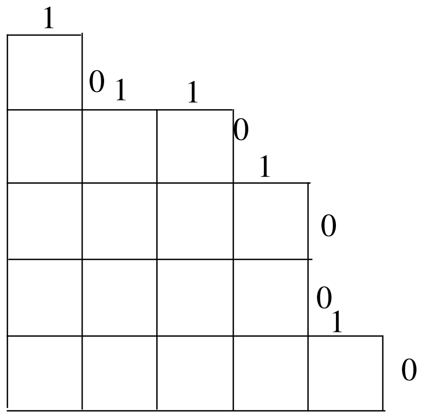
Beweis des zweiten Verfahrens

Zu zeigen ist, daß diese Methode die Zahl m mit den richtigen Wahrscheinlichkeiten in die Ekken des Tableaus bringt, damit ist dann bewiesen, daß die Tableaux gleichverteilt ausgewählt werden.

Partitionen 2

Binärmodell

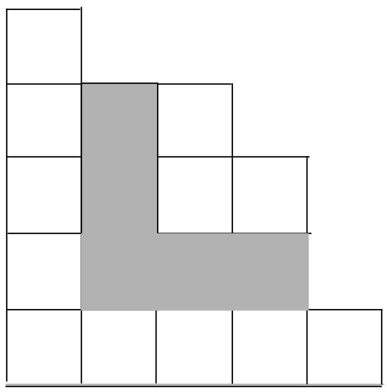
Man kann Partitionen auch als 0-1 Folge, d.h. als Bitvektor abspeichern. Dazu betrachtet man wieder das Ferrers Diagramm, und notiert für jede waagrechten Strich am Rand eine 1und für jeden senkrechten eine 0:



D.h. der Partition 1,3,4,4,5 entspricht die 0-1 Folge 1011010010. Man bemerkt, daß jede 0-1 Folge mit einer 1 beginnt und einer 0 endet. Umgekehrt ist jede 0-1 Folge eine Partition. Die Anzahl der 1'en ist der größte Teil der Partition, die Anzahl der 0'en ist die Anzahl der Teile. Die Konjugation entspricht dem Vertauschen von 0 und 1 und Umdrehen der Leserichtung. Es ist nützlich bei dieser Darstellung auch führende 0'en und endende 1'en zu zulassen. Bei dieser Darstellung ist die Berechnung des Gewichts etwas schwieriger.

Haken

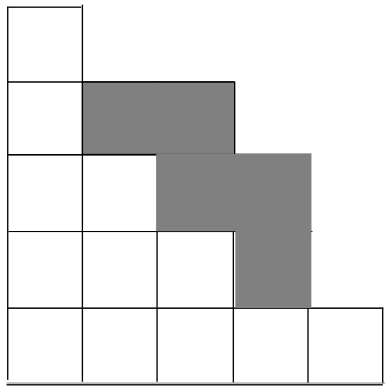
Haken sind spezielle Partitionen 1^k,n, der Name ist klar, wenn man das Diagramm betrachtet. Unter Hakenlänge versteht man das Gewicht der Partition. Zu jeder Box (i,j) in einem Ferrers Diagramm kann man einen Haken $h_{i,j}$ definieren, dies sind die Boxen oberhalb und rechts.



Der Haken zur Box (2,2) wurde schraffiert, die Hakenlänge $|h_{2,2}|$ ist 5. Im folgenden Bild wurde zu allen Boxen die Hakenlänge notiert.

1
4 2 1
6 4 3 1
7 5 4 2
9 7 6 4 1

Zu jedem Haken h gehört der *Randhaken*, dies sind die verbundenen Boxen am Rand des Ferrersdiagramms die vom obersten Kästchen von h bis zum rechtesten Kästchen von h gehen. Im obigen Beispiel:



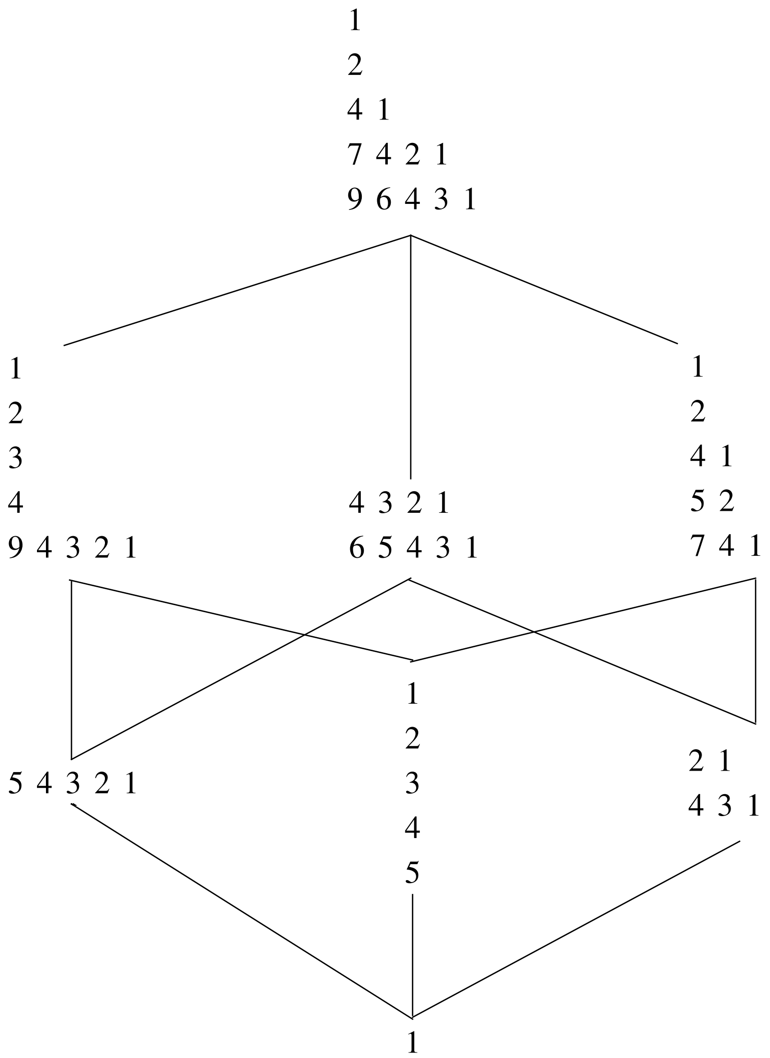
Es handelt sich um eine Bijektion zwischen Haken und Randhaken.

Entfernen von Haken

Eine wichtige Operation ist das Entfernen von Haken aus einer Partition, dies ergibt wieder eine Partition wenn man die übrig gebliebenen Boxen wieder nach links unten packt. Entfernt man z.B. in obiger Partition (1,3,4,4,5) den einzigen Haken der Länge 5, so erhält man die Partition (1,1,2,3,5). Das Entfernen von Haken der Länge k ist sehr leicht wenn man Partitionen als Binärmodell vorliegen hat. Man sucht ein Paar (1,0) wo die 0 exakt k Stellen weiter rechts ist, und ersetzt dann die 1 durch eine 0 und die 0 durch eine 1. Im obigen Beispiel wird aus 1011010010 die Folge 1001010110 was die Partition (1,1,2,3,5) kodiert.

k-Haken Poset

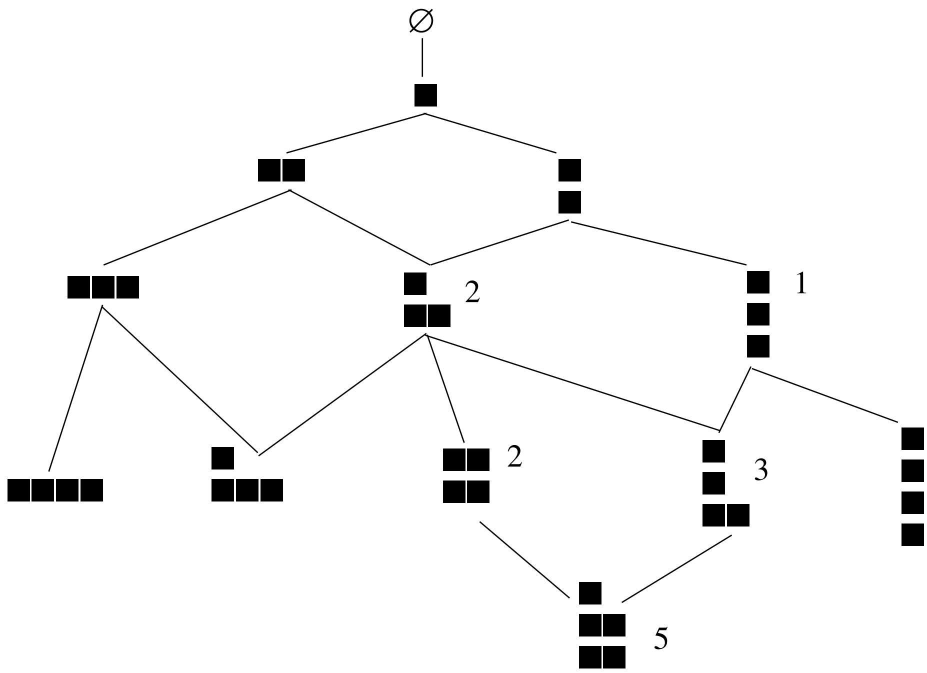
Das Entfernen von k Haken aus einer Partition λ bildet wieder ein Poset $\mathcal{P}_k(\lambda)$. Als Beispiel betrachten wird das Entfernen von 4-Haken aus der Partition (1,1,2,4,5) und es werden die Hakenlängen notiert



Fragen an $\mathcal{P}_k(\lambda)$:

1. existiert ein Minimum?
2. ist dies ein Verband?
3. wieviel Einträge sind in einer Ebene?

Diese Fragen sind klar im Falle k=1, dann ist dies ein endlicher Teilverband des Young Verbands $\mathcal{YP}$, in dem Partitionen benachbart sind, falls sie durch Anfügen/Entfernen einer Box entstehen. Der erste Teil dieses Verbands:



....

Dies ist ein Beispiel eines unendlichen Posets. Die rank Funktion ist das Gewicht der Partition, die Anzahl der Einträge in einer Schicht ist die Anzahl der Partitionen. Eine interessante Frage ist:

Wieviel verschiedene Wege führen zu einer Partition im Young Verband?

Ein Weg wird kodiert durch die Reihenfolge in der die einzelnen Boxen angefügt werden. Man notiert in der Box die Ebene, in der angefügt wurde. Zur Partition λ=(1,2,2) führen insgesamt 5 Wege wie oben notiert, kodiert werden diese durch die 5 Füllungen des Ferrers Diagramm von λ :

$$\begin{array}{ll} 3 & \\ 2 & 5 \\ 1 & 4 \end{array}, \quad \begin{array}{ll} 4 & \\ 2 & 5 \\ 1 & 3 \end{array}, \quad \begin{array}{ll} 5 & \\ 2 & 4 \\ 1 & 3 \end{array}, \quad \begin{array}{ll} 4 & \\ 3 & 5 \\ 1 & 2 \end{array}, \quad \begin{array}{ll} 5 & \\ 3 & 4 \\ 1 & 2 \end{array}$$

Aufgrund der Regel zum Aufbau der Partition durch Anfügen gilt, daß die Einträge in den Zeilen und in den Spalten streng monoton steigen. Ein derartiges Objekt heißt Standard Young Tableau vom Umriß λ. Die Frage nach der Anzahl der Wege ist die Frage nach der Anzahl f_λ der Tableaux in SYT(λ) Der wichtige Satz ist die *Haken Formel* von Frame/Robinson/Thrall der besagt:

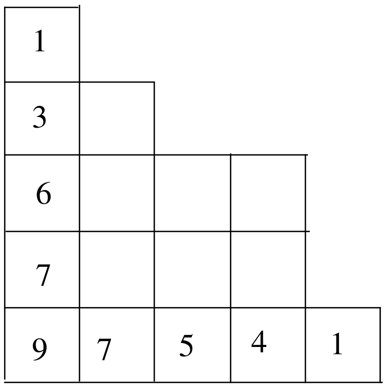
$$f_\lambda = \frac{|\lambda|!}{\prod h_{i,j}}$$

Im obigenBeispiel sind die Hakenlängen

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

man rechnet 5 = 5!/(4·3·2·1·1)

Bemerkung:
Man kann eine Partition auch durch die Länge der Haken in der ersten Reihe oder Zeile beschreiben.



d.h. sowohl die Folge (1,3,6,7,9) als auch die Folge (1,4,5,7,9) kodieren die Partition, man muß jeweils die Folge (0,1,2,3,4) abziehen. Durch die Addition dieser (staircase) Partition wurd die Partition streng monoton steigend. Dies ergibt folgende Gleichheit:

Anzahl der Partitionen von k in verschiedenen Teilen

$p_{strict}(k)$=
Anzahl von k in einen Teil +
Anzahl von k-1 in 2 Teilen +
Anzahl von k-3 in 3 Teilen +
... +
Anzahl von k - (1+2+...+k-1) in k Teilen.

Übungsaufgabe:
Was ist die erzeugende Funktion für die Partitionen in verschiedenen Teilen?

Betrachtet man nun die erzeugende Funktion der Partitionen mit ungeraden Teilen

$$\frac{1}{(1-x)\cdot(1-x^3)\cdot\ldots\cdot(1-x^{2k+1})\cdot\ldots} = \sum p_{odd}(i)x^i$$

und multipliziert diese mit

$$(1-x)\cdot(1-x^2)\cdot(1-x^3)\cdot\ldots$$

so erhält man

$$(1-x^2)\cdot(1-x^4)\cdot(1-x^6)\cdot\ldots$$

Das selbe Ergebnis erhält man aber auch wenn man die erzeugende Funktion für die Partitionen mit verschiedenen Teilen mit

$$(1-x)\cdot(1-x^2)\cdot(1-x^3)\cdot\ldots$$

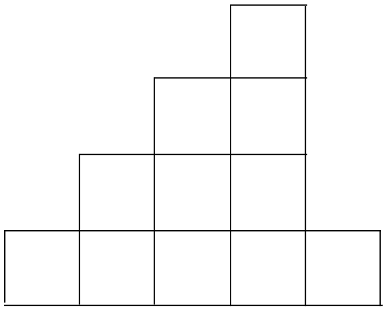
multipliziert, man hat also folgenden

Satz über p_{odd} und p_{strict}

Die Anzahl der Partitionen von n mit verschiedenen Teilen ist gleich der Anzahl der Partitionen von n mit ungeraden Teilen.

Übungsaufgabe:
Finde eine Bijektion zwischen den Partitionen von n in verschiedenen Teilen und den Partitionen von n mit ungeraden Teilen.

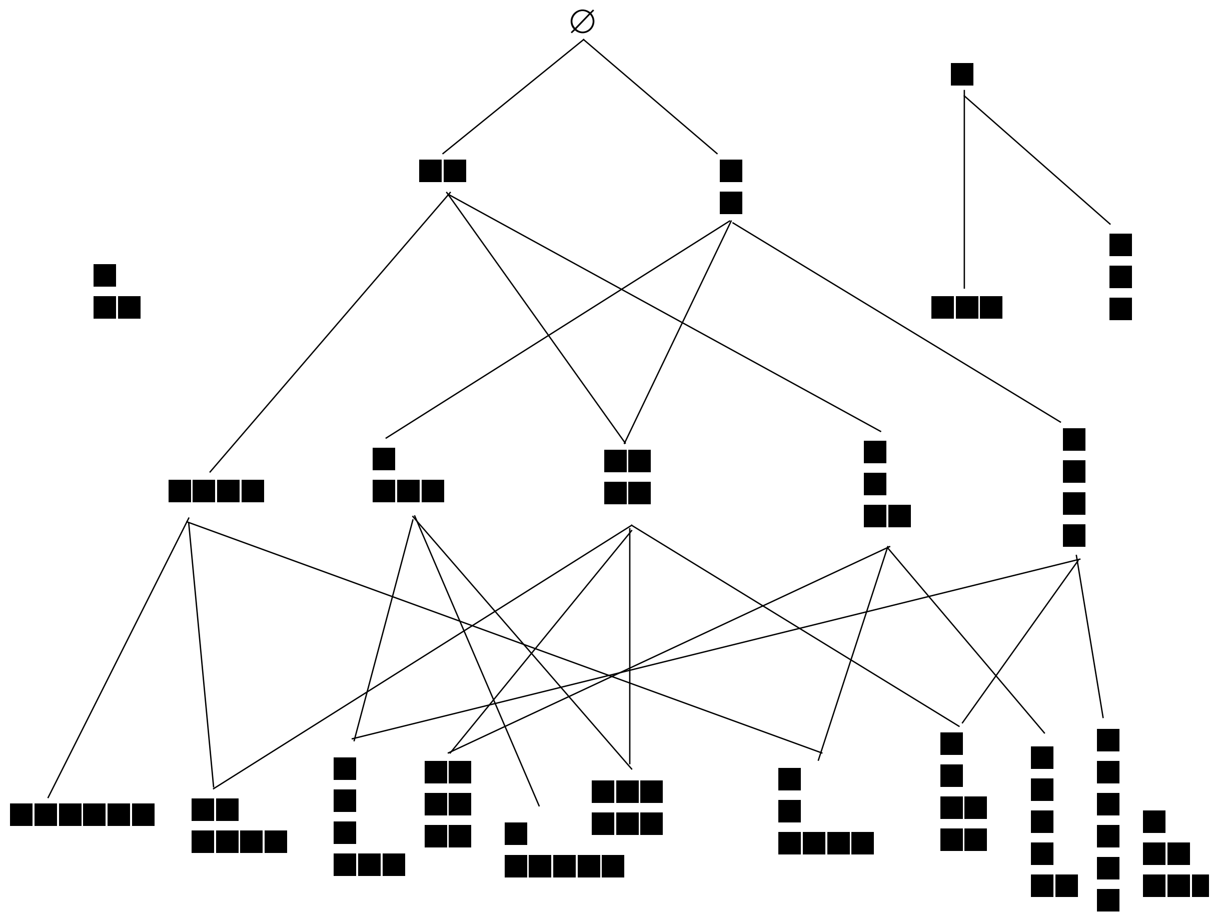
Hat man eine Partition in verschiedenen Teilen, so kann man das *shifted Ferrers Diagramm* definieren:



Die Einträge werden von Zeile zu Zeile eins nach rechts versetzt, obiges Bild ist das shifted Ferrers Diagramm zur Partition (1,2,3,5)

Das k-Haken Poset ist ein endlicher Teil des unendlichen,

k-Haken Young Poset $\mathcal{YP}_k$



Dies ist das Beispiel $\mathcal{YP}_2$. Das Poset $\mathcal{YP}_k$ hat kein Minimum, da es viele Partitionen gibt, die sich nicht weiter durch Entfernen von k-Haken verkleinern lassen, z.B. kann von der Partition (1,2,3) kein 2 Haken entfernt werden. Solche Partitionen heißen *k-irreduzibel.*

Übungsaufgabe:
Finde eine unendliche Famile von Partitionen ohne k-Haken.

Die k-irreduzible Partition, die man aus einer Partition λ, durch entfernen aller k-Haken erhält heißt *k-Kern* der Partition λ. Es gilt der

Satz vom k-Kern

Der k-Kern einer Partition ist eindeutig bestimmt.

Beweis:
Dazu betrachtet man die 0-1 Folge, die den Rahmen der Partition im Binärmodell kodiert. Man schreibt diese 0-1 Folge so, daß die 1-0 Paare die durch Entfernen eines k-Hakens vertauscht werden übereinander stehen. D.h. die 0-1 Folge wird in k Spalten geschrieben, dabei werden die führenden 0‘en und die 1‘en am Ende vergessen. Als Beispiel zum Entfernen von 3-Haken bei

der Partition 1,3,4,4,5, die die 0-1 Folge ..1011010010.. hat wird die 0-1 Folge geschrieben als:

$$\begin{array}{c}101\\101\\001\\011\end{array}$$

Diese Schreibweise ist nicht eindeutig, es ginge auch:

$$\begin{array}{c}010\\110\\100\\101\end{array}$$

Das Entfernen von k-Haken entspricht nun dem nach unten Schieben von 1‘en, dies ist unabhängig unter den Spalten und das Endergebnis ist stets eindeutige 0-1 Folge, die man durch nach unten Schieben aller 1‘en erhält. Dies ist die 0-1 Folge des eindeutig bestimmten k-Kerns.
Im obigen ersten Beispiel ist dies die 0-1 Folge

$$\begin{array}{c}001\\001\\101\\111\end{array}$$

= ...001001101111..., die Partition 1,1,3 die 3-irreduzibel ist, dies ist der 3 Kern der Partition 1,3,4,4,5.

k-Haken 0-1 Folge

Um die obigen Mehrdeutigkeiten in der k-spaltigen Notation zu beseitigen führt man folgende eindeutige Notation ein: Nach den Entfernen der k-Haken soll die erste 1 in der linken Spalten stehen. D.h. sind nach dem Entfernen der Haken in der i-ten Zeile 1‘en , so ist die erste 1‘en in den ersten Spalte. Ansonsten werden führende Zeilen mit 0‘en und Zeilen am Ende mit 1‘en normalerweise nicht gezeichnet. Dies ist die *k-Haken 0-1 Folge* der Partition. Im obigen Beispiel wäre die zweite Möglichkeit

$$\begin{array}{c}010\\110\\100\\101\end{array}$$

richtig gewesen, da dies nach dem Entfernen die 0-1 Folge

$$\begin{array}{c}000\\100\\110\\111\end{array}$$

geliefert hätte, in der die 1‘en in der zweiten Zeile alle links sind. Die Rechtfertigung dieser Notation ergibt sich auch aus dem Fall des leeren k-Kerns, da dann nur Zeilen mit 0‘en und 1‘en übrig bleiben.

Anzahl der Schichten im k-Haken Young Poset $\mathcal{YP}_k$

Im obigen Beispiel der 2 Haken, betrachtet man den Teilverband ausgehend von der leeren Par-

tition und man betrachtet die Zahlen
$p_2(i,\emptyset)$ = Anzahl der Partition von 2*i mit leeren 2-Kern
$p_2(i,1)$ = Anzahl der Partition von 2*i+1 mit 2-Kern=1
$p_2(i,12)$ = Anzahl der Partition von 2*i+3 mit 2-Kern=12
....
und man bemerkt. daß diese Zahlenfolgen identisch sind:

i	$p_2(i)$	i	$p_2(i)$
0	1	6	65
1	2	7	110
2	5	8	185
3	10	9	300
4	20	10	481
5	36	11	752

Dazu betrachtet man als erstes die Anzahl der Elemente in einer Schicht z.B. des Teilverbands mit leeren k-Kern. In der Schicht i sind die Partition, die durch Entfernen von i k-Haken die leere Partition liefern. Betrachtet man die k-Haken 0-1 Folge, so wurde i mal eine 1 mit einer darunterliegenden 0 vertauscht. Betrachtet man die 0'en und 1'en in einer Spalte der k-Haken 0-1Folge, so erhält man wenn man von oben nach unten liest eine 0-1 Folge, wobei zuerst 0'en kommen, dann machmal 0'en und wieder 1'en und am Schluß lauter 1'en. So eine Folge kann man wieder als Partition lesen, und das Gewicht gibt gerade die Anzahl der Haken, die in dieser Spalte entfernt werden können. Als Beispiel betrachtet man die 3-Haken 0-1 Folge:

010
011
100
101

Dies liefert die 3 0-1 Folge ....0011..., ...1100..., ...0101.... Dies sind die 3 Partitionen $(\emptyset,(2,2),(1))$. In diesem Beispiel werden also 5 Haken entfernt um zum leeren 3 Kern zu kommen. Dies funktioniert auch bei nicht leeren k-Kern, da man in den Spalten am Ende stets die leere Partition erhält. Damit hat man folgenden Satz bewiesen:

Satz von der Anzahl der Partitionen mit festen k-Kern

Die Anzahl $p_k(i,\lambda)$ der Partitionen vom Gewicht $k*i+|\lambda|$ mit k-Kern λ ist

$$\sum_{i_1+i_2+\ldots+i_k=i} p(i_1)\cdot p(i_2)\cdot\ldots\cdot p(i_k)$$

wobei p(n) wie üblich die Anzahl der Partitionen von n bezeichnet. Wobei p(0)=1. Man sieht, daß die Anzahl wie schon beobachtet unabhängig vom k-Kern ist.

Obige Tabelle der $p_2(i,\emptyset)$ gibt also die Anzahl der Möglichkeiten die Zahl i als Summe von 2 Partitionen zu schreiben. D.h. $p_2(6,\emptyset)$= 2*p(0)*p(6)+2*p(1) * p(5) + 2*p(2)*p(4) + p(3)*p(3) = 2*11+ 2 * 7 + 2 * 2*5 + 3 * 3 = 22 + 14 + 20 + 9 = 65.

Das k-Tupel von Partitionen, welches man durch spaltenweises Lesen der k-Haken 0-1 Folge einer Partition λ erhält wird als *k-Quotient* der Partition λ bezeichnet. Das Gewicht des k-Quotienten ist die Summe über die Gewichte der k Partitionen. Obige Methode liefert eine Bijektion zwischen den Partitionen vom Gewicht $k*i+|\lambda|$ mit k-Kern λ und den k-Quotienten vom Gewicht i.

Beispiel zur Bijektion zwischen k-Quotient und Partitionen mit festen k-Kern

Man betrachte den 4-Quotienten ($\emptyset$, (1), (1,2), (3)) und den den 4-Kern (2,5). Der 4-Kern hat als 0-1 Folge ....0000011011101111...... Der 4-Kern hat folgende 4 Haken 0-1 Folge

0000
0000
1101
1101

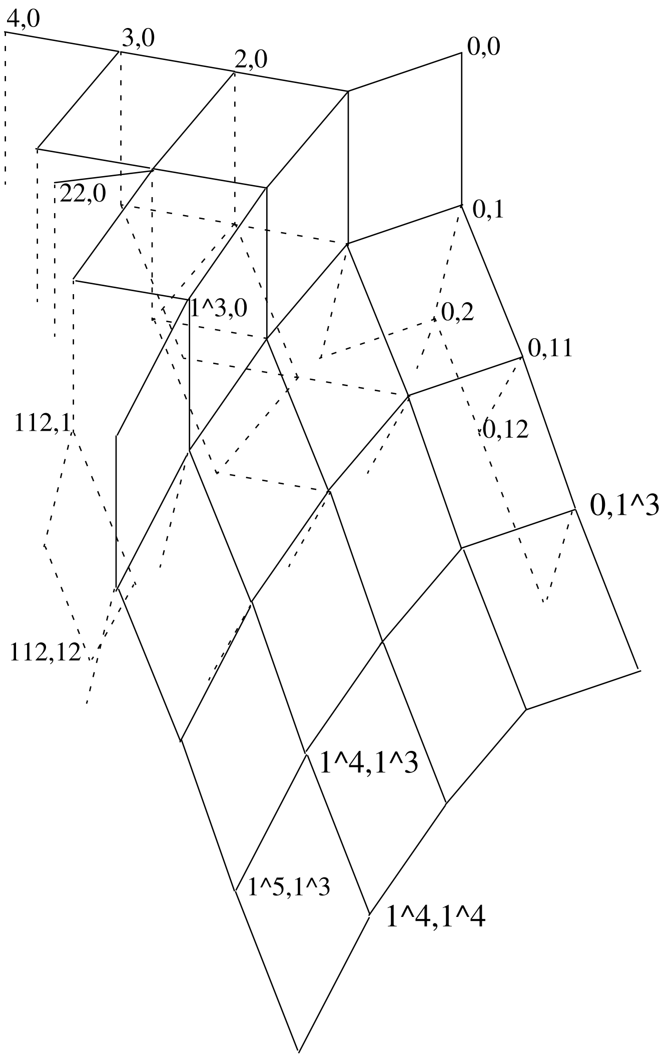
Die 4 Partitionen haben folgende 0-1 Folgen:
$\emptyset$ = ...0000001111111..., d.h. die erste Spalte wird nicht mehr geändert.
(1) = ...0000001011111.., in der zweiten Spalte wird aus 0011 die Folge 0101.
(1,2) = ...0001010111.., d.h. es müssen noch zwei Zeilen mit 1‘en angehängt werden und aus 000011 wird 001010
(3) = ...010001111.., es muß noch eine Zeile mit 0‘en am Anfang betrachtet werden und aus 0001111 wird 1000111. Die gesuchte Partition aus der insgesamt 7 4-Haken entfern werden um den 4-Kern (2,5) zu erhalten ist dann

0001
0000
0100
1010
1101
1111
1101

Insgesamt ergibt dies die 0-1 Folge 0001 0000 0100 1010 1101 1111 1101, dies kodiert die Partition (1,1,1,1,1,2,2,3,4,6,13) vom Gewicht 35 = 7+7*4.

Nachdem man diese Bijektion gefunden hat, wird klar, daß man das Poset besser anders visualisiert, so z.B. das Teilposet von $\mathcal{YP}_2$ welches die Partitionen mit leeren 2-Kern enthält. Man kann sich prinzipiell auf ein Teilposet mit fest k-Kern zurückziehen, da die anderen Posets zu anderen Kern wie gesehen isomorph sind. Dieses Poset zu einem festen k-Kern wird mit $\mathcal{HP}_k$ bezeichnet.

Beispiel $\mathcal{HP}_2$



Ein Knoten wird durch ein Paar zweier Partitionen (λ,μ) beschreiben, dieses Paar ist in Bijektion zu einer Partition vom Gewicht 2*(|λ|+|μ|) mit leeren 2-Kern. Im Poset wurde als Beschriftung dieses Paar gewählt. Es wurde nur ein Teil gemalt. Durch die Beschreibung der Partitionen mittels k-Quotient sieht man auch sofort:

Satz

$\mathcal{HP}_k$ ist ein Verband

Beweis: das Supremum und das Infimum sind die komponentenweisen Suprema und Infima. Es handelt sich bei $\mathcal{HP}_k$ um das k-fache direkte Produkt des Young Verbands, und natürlich ist das direkte Produkt eines Verbands wieder ein Verband..

erzeugende Funktion

$$\prod \frac{1}{(1-x^i)^k} = \sum p_k(i) x^i$$

Permutationen III

längste ansteigende Teilfolge

Gegeben seien n verschiedene Zahlen $(k_1,....,k_n)$ in einer beliebigen Reihenfolge, und man will die Länge und auch die Zahlen der längsten ansteigenden Teilfolge. Diese Fragestellung veranlaßte Schensted folgenden nach ihm benannten Algorithmus vorzustellen. Als Eingabe dient eine Permutation $\pi=[\pi_1,...,\pi_n]$, und als Ergebnis erhält man zwei Standard Young Tableau P,Q von gleichem Umriß.

Dazu definiert man einen Algorithmus, der in ein Standard Tableau t vom Gewicht eine Zahl k einfügt. Das Ergebnis ist ein Standard Tableau vom Gewicht $|t|+1$, welches zusätzlich die Zahl k enthält, dieses Tableaux wird mit $k \to t$ bezeichnet.

Zeilen - Bumping - Algorithmus

EINGABE:	Standard Tableau t, Zahl k
AUSGABE:	Standard Tableau $k \to t$
ALGORITHMUS:	B1: i =1 B2: existiert keine i-te Zeile, so wird k der erste Eintrag in der i-ten Zeile; ENDE B3: Suche in der i-ten Zeile die kleinste Zahl m > k B4: existiert sie nicht hänge k an die i-te Zeile; ENDE B5: ersetze m durch k; i++; k=m; goto B2;

Beispiel: Es soll $3 \to \begin{matrix} 7 & 9 \\ 2 & 5 \end{matrix}$ berechnet werden. In der ersten Zeile ist m=5, die 5 wird durch 3 ersetzt und nun soll die 5 in die zweite Zeile gebracht werden, sie ersetzt die 7 und die 7 soll in die dritte Zeile, die gibt es noch nicht und so wird die 7 der erste Eintrag in der 3. Zeile. Das Ergebnis ist:

$$\begin{matrix} 7 & \\ 5 & 9 \\ 2 & 3 \end{matrix}$$

Der Schensted Algorithmus besteht nun darin mit dem leeren Tableau zu beginnen und nach einander π_1 , π_2, ..., π_n einzufügen. Dies ergibt das Tableau P. D.h.

$$P = \pi_n \to ... \to \pi_1 \to \emptyset$$

Bei diesem Algorithmus wird in jedem Schritt genau ein weiteres Kästchen an das Ferrers Diagramm angehängt (B2 oder B4), diese Beobachtung dient zur Definition des Q Tableau, in das Kästchen welches im i-ten Schritt angefügt wird, schreibt man beim Q Tableau die Zahl i. Das Q Tableau notiert die Reihenfolge des Aufbaus des P Tableau.

Beispiel Schensted Algorithmus

Die Permutation [4,1,5,3,2]

P	Q
∅	∅
4	1
4 1	2 1
4 1 5	2 1 3
4 5 1 3	2 4 1 3
4 3 5 1 2	5 2 4 1 3

Der Algorithmus liefert eine Abbildung, die einer Permutation zwei SYT vom gleichen Umriß zuordnet. Es gilt folgender

Satz über die Schensted Bijektion

1. Der Algorithmus von Schensted liefert eine Bijektion zwischen den Permutation π vom Grad n und Paaren (P,Q) von Standard Young Tableaux vom Gewicht n und gleichen Umriß.
2. Der Permutation π^{-1} wird unter dieser Bijektion das Paar (Q,P) zugeordnet.

Dazu nochmal ein Beispiel, die Permutation [2,5,4,1,3]

P	Q
∅	∅
2	1
2 5	1 2
5 2 4	3 1 2
5 2 1 4	4 3 1 2
5 2 4 1 3	4 3 5 1 2

Satz über die Anzahl der Involutionen

Aus obigen Satz erhält man als Folgerung folgenden Satz:

Die Anzahl der Involutionen in der S_n ist die Anzahl der SYT vom Gewicht n

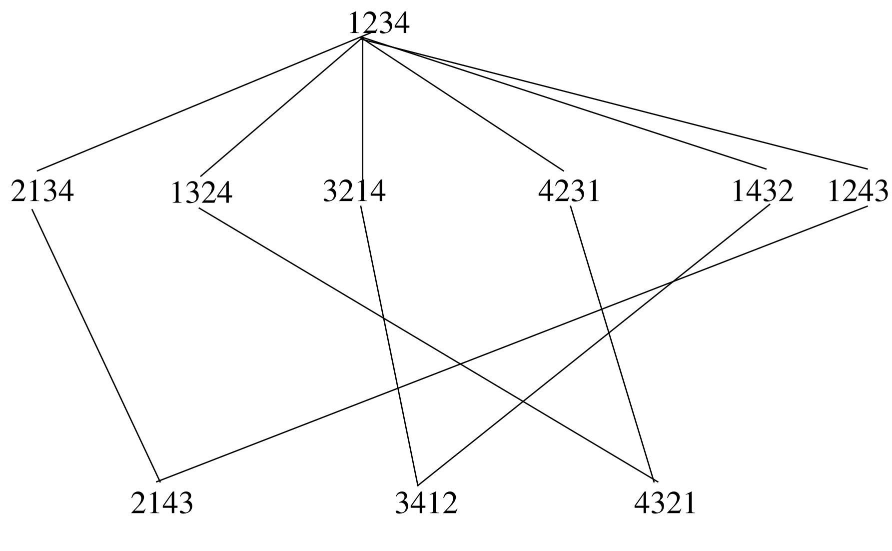
Die Involutionen sind gerade die Permutationen der S_n, die unter der Schensted Bijektion auf ein Paar (t,t) abgebildet werden, d.h. wo P und Q Symbol identisch sind.

Beispiel zur Anzahl der Involutionen

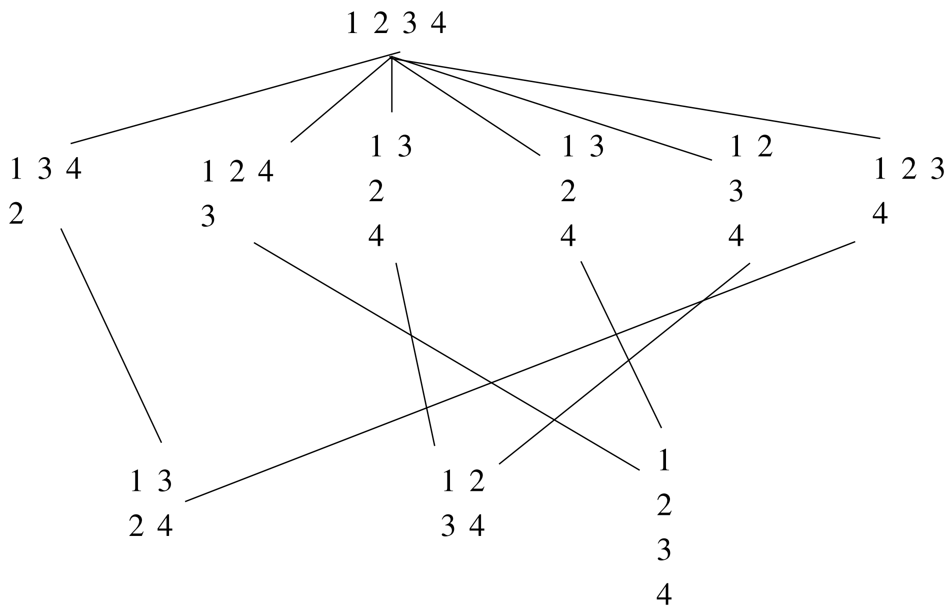
Im Falle der S_4 gibt es 10 Involutionen

[1,2,3,4]	[2,1,3,4]	[3,2,1,4]	[4,2,3,1]	[1,3,2,4]
1 2 3 4	1 3 4 2	1 4 2 3	1 3 2 4	1 2 4 3
[1,4,3,2]	[1,2,4,3]	[2,1,4,3]	[3,4,1,2]	[4,3,2,1]
1 2 3 4	1 2 3 4	1 3 2 4	1 2 3 4	1 2 3 4

Dies kann man auch wieder als Poset visualisieren:



Die Anzahl der Fixpunkte gibt die Schicht in der eine Involution liegt. Benachbart sind Involutionen wenn sie durch eine Transposition unterscheiden. Malt man das Poset mit Tableaux statt mit Permutationen:



so fällt auf, daß Tableaux mit gleichem Umriß in der gleichen Schicht liegen. Es gilt folgender

Satz [Schützenberger]

Zu Involutionen mit k Fixpunkten gehören die Tableaux mit k Spalten ungerader Länge

Satz über die längste ansteigende Teilfolge

Nach dem Schensted Algorithmus ist die länge der erste Zeile des Tableau P, die Länge der längsten ansteigenden Teilfolge in der Ausgangspermutation, und die Länge der ersten Spalte die Länge der längsten absteigenden Teilfolge in der Ausgangspermutation.

Major Index

Stirling Zahlen

Euler Zahlen

Teilmengen I

alle Teilmengen einer endlichen Menge

Gegeben sei eine Menge M mit n verschiedenen Elementen. Man kodiert Teilmengen durch Bitvektoren der Länge n, ist das i-te Bit gesetzt, so bedeutet dies, das i-te Element ist in der Teilmenge.

Beispiel:
Gegeben sei die Menge M = {a,b,c,d,e,f,g,h}, dann kodiert der Bitvektor 01001101 die Teilmenge {b,e,f,h}.

Anzahl

Es gibt insgesamt 2^n Teilmengen, liest man die Bitvektoren der Länge n als Zahlen in Binärschreibweise sind dies die Zahlen 0 bis 2^n -1.

random / next

durch die Bijektion zwischen Teilmengen und den Zahlen, kann man auch leicht die nächste Teilmenge und auch leicht eine zufällige Teilmenge auswählen.

Gray Code [Wilf]

Obige Kodierung der Teilmengen hat den Nachteil, daß sich z.B. die 31-te und 32-te Teilmenge stark unterscheiden. Dies ist ein Problem wenn man z.B. bei der Übertragung der Information über einen analogen Kanal mit Fehlern rechnen muß. Die prinzipielle Fragestellung war, wie finde ich eine Reihenfolge der Bitstrings, sodaß sich zwei benachbarte nur gering unterscheiden, oder noch besser, daß sich zwei benachbarte nur um ein Bit unterscheiden. Dies funktioniert mit dem *Gray Code*.

Beispiel

Binär	Gray	Binär	Gray
0000	0000	1000	1100
0001	0001	1001	1101
0010	0011	1010	1111
0011	0010	1011	1110
0100	0110	1100	1010
0101	0111	1101	1011
0110	0101	1110	1001
0111	0100	1111	1000

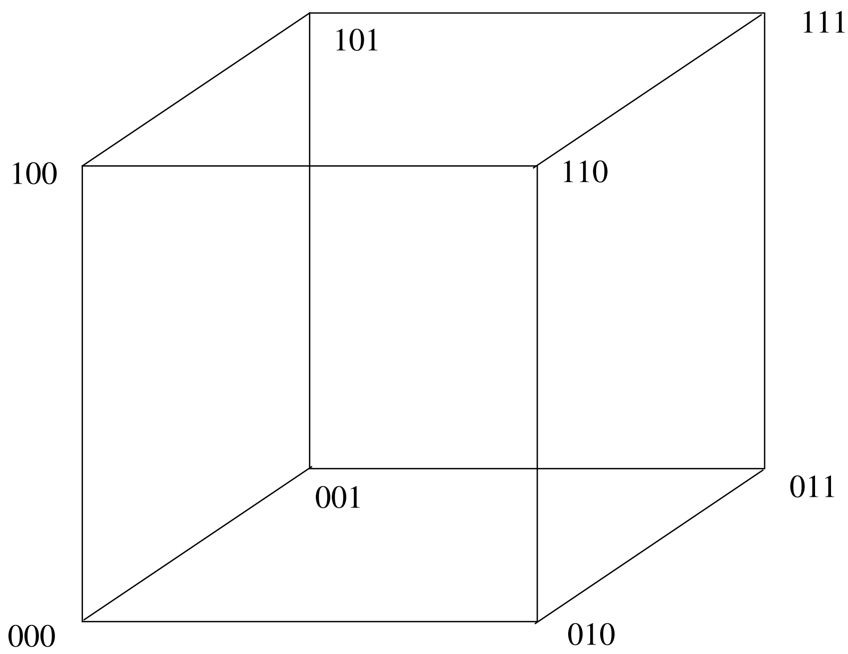
Dies ist der 4-stellige Gray Code $\mathcal{L}_4$.

Algorithmus zur Erzeugung des Gray Codes

Um den allgemeinen Gray Code $\mathcal{L}_n$ zu erzeugen, verwendet man folgende einfache rekursive Regel.

1. $\mathcal{L}_0$ ist die leere Liste
2. für n>0 nimmt man einmal die Liste $\mathcal{L}_{n-1}$ wobei jeder Eintrag ein führendes Bit 0 bekommt und noch eine Kopie von $\mathcal{L}_{n-1}$ in umgekehrter Reihenfolge wobei jeder Eintrag ein führendes Bit 1 bekommt, dies ist dann $\mathcal{L}_n$.

Auf diese Weise wurde auch obiger Code$\mathcal{L}_4$ erzeugt. Der erzeugt Code, ist der *standard reflected Gray code*. Man beweist durch Induktion, daß benachbarte bit strings sich nur durch ein Bit unterscheiden. Für den Zweck der Teilmengen, kann man mittels Gray Code die Teilmengen so durchlaufen, daß sich zwei benachbarte Teilmengen nur um ein Element, daß hinzu kommt oder weg kommt, unterscheiden. Dies ist ein Durchlauf, der ähnlich dem Trotter[Seite 11] Durchlauf der Permutationen ist. Man kann die Bitstrings des Gray Codes auch als Koordinaten eines n-dimensionalen Würfels betrachten.



Ein Gray Code ist dann ein Hamiltonweg durch den n dimensionalen Würfel, denn Kanten bestehen genau zwischen Bitstrings, die sich um ein Bit unterscheiden.

offenes Problem

Wieviel Hamiltonwege gibt es in einem n dimensionalen Würfel, d.h. wieviele Gray Codes gibt es?

nächste Objekt im Gray Code

Man betrachtet den Index j des Bits welches beim Übergang zur nächsten Gray Code Bitfolge geändert wird. Im Beispiel von $\mathcal{L}_4$ ist die Folge:

121312141213121

Man beweist folgende Regel:
Sei k die Anzahl der 1'en im Code, der Index im Bitvektor, an dem geändert wird ist 1 wenn k gerade ist. Ist k ungerade wird nach der ersten 1 im Bitvektor geändert.

Es empfiehlt sich die Variable k zu merken, wenn man die Objekte des Gray Codes durchlaufen will. Der letzte Code ist erreicht, wenn k ungerade ist und die erste 1 an der letzten Stelle steht.

EINGABE:	Cray Code $c=c_1...c_n$
AUSGABE:	nächster Cray Code $c=c_1 \ldots c_n$
ALGORITHMUS:	C1: sei k die Anzahl der 1'en C2: ist k gerade setze j=1 goto C5 C3: sei j der kleinste Index mit c_j=1 C4: ist j gleich n dann ist der letzte Cray Code C5: invertiere das j-te Bit

rank und unrank für den Gray Code

Es gilt folgender

Satz über das Unranking beim Gray Code[Wilf]

Sei $m = \sum \varepsilon_i 2^i$ die Binärschreibweise einer Zahl m, und sei $\ldots e_2\ e_1\ e_0$ der Bitstring des m-ten Gray Codes, dann gilt für i = 0,1,..:

$$e_i \equiv \varepsilon_i + \varepsilon_{i+1} \pmod 2$$

Beweis:
Man beweist dies per Induktion nach n. Es ist klar für n = 0,1. Der Satz gelte für die Gray Codes aus $\mathcal{L}_{n-1}$. Sei nun m der Rank eines Strings aus $\mathcal{L}_n$.
Ist $m < 2^{n-1}$, so ist nichts zu beweisen, denn es ist der Bitstring aus $\mathcal{L}_{n-1}$ mit einer zusätzlichen führenden 0. Diese erhält man auch nach obiger Regel, da $\varepsilon_n + \varepsilon_{n+1} = 0+0= 0$.
Ist $m > 2^{n-1}$ -1, dann betrachte $m' = 2^n - 1 - m$, dann ist $m' < 2^{n-1}$ und nach Induktionsannahme kann man nach der Regel den Bitstring für m' berechnen. Man weiß daß sich der Bitstring des Gray Codes von m und m' nur dadurch unterscheiden, daß an der n-ten Stelle bei m' eine 0 steht und bei m eine 1. Die Binärdarstellung von m und m' sind genau komplementär, d.h. betrachtet man die n-stellige Darstellung von m und m', so ist m das 2er Komplement von m' und umgekehrt, wobei bei m an der n-ten Stelle eine 1 steht. Man prüft nun die Gültigkeit obiger Regl nach.

Beispiel zum Unranking beim Gray Code

Was ist die 3242442341-te Teilmenge? Dazu schreibt man die Zahl in Binärschreibweise: $3242442341 = 2^{31}+2^{30}+2^{24}+2^{22}+2^{17}+2^{16}+2^{15}+2^{13}+2^{12}+2^{11}+2^{10}+2^{9}+2^{6}+2^{5}+2^{2}+ 2^{0}$ Dies ergibt die Binärdarstellung: ..01100000101000011101111100110010 1. Man betrachtet nun die Paare in der Binärdarstellung und bildet die 0-1 Folge des Gray Codes nach der Regel:

$$e_i \equiv \varepsilon_i + \varepsilon_{i+1} \pmod 2$$

man erhält:

..01100000101000011101111100110010 1
..01010000111100010011000010101011 1

zur besseren Übersichtlichkeit wurden die Stellen im Gray Code direkt unter die Stellen der Binärdarstellung geschrieben. Die 3242442341-te Teilmenge (man beginnt mit der 0-ten Menge) besteht aus den Elementen

{1,2,3,5,7,9,14,15,18,22,23,24,25,30,32}

wenn man mit dem Elementen 1,2,3,... arbeitet.

Satz über das Ranking beim Gray Code[Wilf]

Sei $m = \sum \varepsilon_i 2^i$ die Binärschreibweise einer Zahl m, und sei $...e_2\ e_1\ e_0$ der Bitstring des m-ten Gray Codes, dann gilt für i = 0,1,..:

$$\varepsilon_i \equiv e_i + e_{i+1} + e_{i+2}+ e_{i+3}+ ... \pmod 2$$

Beweis: aus dem Satz über das Unranking.

Beispiel zum Ranking beim Gray Code

Die wievielte Teilmenge ist die Teilmenge {1,3,5,7,9,11}? Die Teilmenge wird durch die die 0-1 Folge ..0010101010101 kodiert.Die Anwendung der Regel

$$\varepsilon_i \equiv e_i + e_{i+1} + e_{i+2}+ e_{i+3}+ ... \pmod 2$$

ergibt die Binärdarstellung

..0010101010101
..0011001100110

dies ist die Zahl $2^1+2^2+2^5+2^6+2^9+2^{10} = 1638$. Die Teilmenge {1,3,5,7,9,11} ist also die 1638-te Teilmenge im Gray Code.

alle k-elementigen Teilmengen einer endlichen Menge

Gegeben sei eine Menge M mit n verschiedenen Elementen, man interessiert sich nun für alle Teilmengen mit k Elementen. Die Anzahl ist $\binom{n}{k}$. Gesucht sind nun Routinen zum Durchlauf und zur zufälligen Erzeugung dieser Objekte. Auch in diesem Fall suchen wir eine Gray Code Methode, zwei Teilmengen die im Durchlauf nacheinander kommen, sollen sich nur durch Ersetzen eines Elements unterscheiden.

revolving door

Der zugehörige Algorithmus heißt aus naheliegenden Gründen *revolving door* (Drehtür) Algorithmus [Nijenhuis Wilf]. Der Algorithmus läßt sich am einfachsten rekursiv definieren. Sei dazu A(n,k) die Liste der k-elementigen Teilmengen der n-elementigen Grundmenge, dabei sei die erste Teilmenge {1,2,...,k} und die letzte {1,2,...,k-1,n}. Man verwendet folgende Rekursion:

1. A(k,k) = {1,2,...,k}; A(k,0) = ∅
2. A(n,k) = A(n-1, k) und die umgekehrte Liste von A(n-1,k-1) wobei den einzelnen Teilmengen das n-te Element hinzugefügt wird. Dies gilt falls 0<k<n.

Man bemerkt, daß diese die gleiche Rekursion ist, wie beim Pascal‘ schen Dreieck. Im Rekursionsschritt kommt nach der Teilmenge {1,2,...,k-1,n-1} die Teilmenge {1,2,....,k-2,n-1,n}, d.h. das Element k-1 wird durch n ersetzt, sodaß der Algorithmus auch wirklich ein revolving door Algorithmus ist, und man vergewissert sich auch, daß die erste und letzte Teilmenge die richtige ist. Die Rekursion läßt sich in Kurznotation schreiben als

$$A(n,k) = A(n-1,k),\ \overline{A(n-1,k-1)}+n$$

dabei bezeichnet der Strich das Umdrehen der Reihenfolge. Dies ist eine allgemeine Methode einen Gray Code zu bilden, im Falle der allgemeinen Teilmengen war die Regel

$$\mathcal{L}_n = 0+\mathcal{L}_{n-1},\ 1+\overline{\mathcal{L}}_{n-1}$$

Beispiel A(6,4)

A(6,4)	A(5,4)	A(4,4)				1234
		$\overline{A(4,3)}$5	A(3,2)45	A(2,2)45		1245
				$\overline{A(2,1)}$345	A(2,0)2345	2345
					$\overline{A(1,1)}$345	1345
			$\overline{A(3,3)}$5			1235
	$\overline{A(5,3)}$6	A(4,2)56	A(3,2)56	A(2,2)56		1256
				$\overline{A(2,1)}$356	A(2,0)2356	2356
					$\overline{A(1,1)}$356	1356
			$\overline{A(3,1)}$456	A(3,0)3456		3456
				$\overline{A(2,1)}$456	A(2,0)2456	2456
					$\overline{A(1,1)}$456	1456
		$\overline{A(4,3)}$6	A(3,2)46	A(2,2)46		1246
				$\overline{A(2,1)}$346	A(2,0)2346	2346
					$\overline{A(1,1)}$346	1346
			$\overline{A(3,3)}$6			1236

Um alle k-elementigen Teilmengen in der revolving door Reihenfolge zu durchlaufen benötigt man eine Regel, die die nächste k-elementige Teilmenge der n-elementigen Grundmenge liefert.

Beispiel zu next beim revolving door Algorithmus
gesucht ist der Nachfolger von C={4,5,7,8} in A(9,4), dies sei kurz mit N(4578) bezeichnet. Es gilt A(9,4) = A(8,4), $\overline{A(8,3)}$+9. Die 9 ist nicht in C, also wird N(C) in A(8,4) =A(7,4), $\overline{A(7,3)}$+8 gesucht. C liegt im zweiten Teil, der wurde umgedreht, gesucht ist jetzt der Vorgänger von {4,5,7} (= P(457)) in A(7,3)=A(6,3),$\overline{A(6,2)}$+7. Der zweite Teil ist relevant, gesucht ist N(45) in A(6,2) = A(5,2), A(5,1)+6, also N(45) in A(5,2) = A(4,2) , $\overline{A(4,1)}$+5, gesucht wird daher P(4) in A(4,1) = A(3,1), $\overline{A(3,0)}$+4. Eigentlich sind wir nun in dem zweiten Teil und suchen den Vorgänger, da dieser nicht existiert, muß man den letzten Eintrag von A(3,1) nehmen, dies ist die 3. Insgesamt ist also der Nachfolger von C die Teilmenge {3,5,7,8}.

Allgemein sucht man $N(s_1,...,s_k)$ in A(n,k) dies wird rekursiv zum Problem der Suche nach $N(s_1,...,s_j)$ oder $P(s_1,...,s_j)$ in A(m,j). Irgendwann liegt der Nachfolger oder der Vorgänger der gesucht wird im anderen Teil der beiden Teile bei der Rekursion. An der Stelle ist man fertig, da das erste bzw. letzte Element in A(m,j) eindeutig bestimmt ist.

$$\{1,2,...j-1,m-1\} \quad \underset{P}{\overset{N}{\rightleftarrows}} \quad \{1,2,...,j-2,m-1,m\} \in A(m,j)$$

Aufgabe eines Algorithmus ist es daher möglichst schnell m und j zu finden. Dazu geht man einfacher von links nach rechts vor, man beginnt mit j = 1 und sucht das kleinste j sodaß $N(s_1,...,s_k)$ bzw $P(s_1,...,s_k)$ existieren. Ob man nach N oder P suchen muß erkennt man aus der Parität von k.

EINGABE:	n, k, k-elementige Teilmenge s_1 ... s_k aus {1,...,n}
AUSGABE:	nächste k-elementige Teilmenge s_1 ... s_k in der revolving door Reihenfolge oder die Rückgabe LAST wenn kein Nachfolger existiert
ALGORITHMUS:	RD1: j = 0 RD2: wenn k gerade gehe nach RD_V sonst nach RD_N RD_N: j++;wenn j=k ist dann setze m=n, sonst setze $m=s_{j+1}$ -1 RD_N2: wenn $m=s_j$ gehe nach RD_V RD_N3: $s_j = s_j+1$; wenn j>1 dann setze $s_{j-1}=s_j-1$; ENDE [ein Nachfolger existiert] RD_V: j++;wenn j>k return LAST RD_V2: wenn $s_j = j$ gehe nach RD_N RD_V3: $s_j = s_j-1$; wenn j>1 dann setze $s_{j-1}=j-1$; ENDE [ein Vorgänger existiert]

zufällige k - Teilmengen

Um eine k elementige Teilmenge einer n elementigen Teilmenge zu erzeugen verwendet man folgenden Trick, in einem Hilfsfeld H speichert man die Zahlen 1 bis n in geordneter Reihenfolge, d.h. $H_i = i$. Man erzeugt nun eine Zufallszahl r_1 zwischen 1 und n, der r_1 -te Eintrag aus H ist der erste Eintrag in der zufälligen Teilmenge, in H tauschen nun der $n-r_1$ -te Eintrag und der n-te Eintrag ihren Platz. Im zweiten Schritt wird eine Zufallszahl r_2 zwischen 1 und n-1 erzeugt, der r_2 -te Eintrag aus H ist der zweite Eintrag in der zufälligen Teilmenge, in H tauschen nun der $n-r_2$ -te Eintrag und der n-1-te Eintrag ihren Platz. Auf diese Weise stehen in den ersten Plätzen von H die möglichen Kandidaten um ein weiteres Element in die k-elementige Teilmenge einzufügen.
Nachteil: ist n groß und k klein so benötigt man viel Speicherplatz für das Hilfsfeld, man sucht daher ein Verfahren mit O(k) Speicherplatz. Ferner liefert obiges Verfahren keine sortierte Folge der Elemente in der k-elementigen Teilmenge.

strong revolving door

Es gibt noch strengere Bedingungen an die Eigenschaft benachbart in der revolving door Reihenfolge zu sein. Betrachtet man die Darstellung der Teilmengen als sortierte Liste der Elemente, so sollen zwei benachbarte sich nur durch einen Eintrag unterscheiden, dies ist die *strong revolving door* Reihenfolge. So sind z.B. die Teilmengen 1235 und 1256 in der revolving door Relation benachbart, nicht aber in der strong revolving door Relation. Man kann diese Relation auch in der Binärdarstellung beschreiben, dann darf eine 1 nur mit 0‘en tauschen zu denen sie kommt ohne über eine weitere 1 zu springen. Hier ein Beispiel für die 3 Teilmengen einer 6 elementigen Teilmenge.

123	234
124	235
125	236
126	256
156	246
146	245
145	345
135	346
136	356
134	456

Die allgemeine Regel um die Liste B(n,k) zu bilden ist:

$$B(n,k) = B(n-2,k-2)+\{n-1,n\}\ ,\ \overline{B(n-2,k-1)}+\{n\},\ B(n-1,k)$$

Diese Reihenfolge wurde 1982 von Brendan McKay and Peter Eades gefunden.

adjacent revolving door

Man kann noch strengere Voraussetzungen an die Nachbarschaftsrelation zweier Teilmengen stellen. Nun sollen nur solche benachbart sein, die durch Ersetzen eines Elements durch ein benachbartes auseinander hervorgehen. Dies sei die *adjacent revolving door* Relation. In der Binärdarstellung bedeutet dieses, daß benachbarte 1-0 Paare tauschen. Dafür gibt es jedoch keinen Gray Code mehr, was folgender Satz beweist:

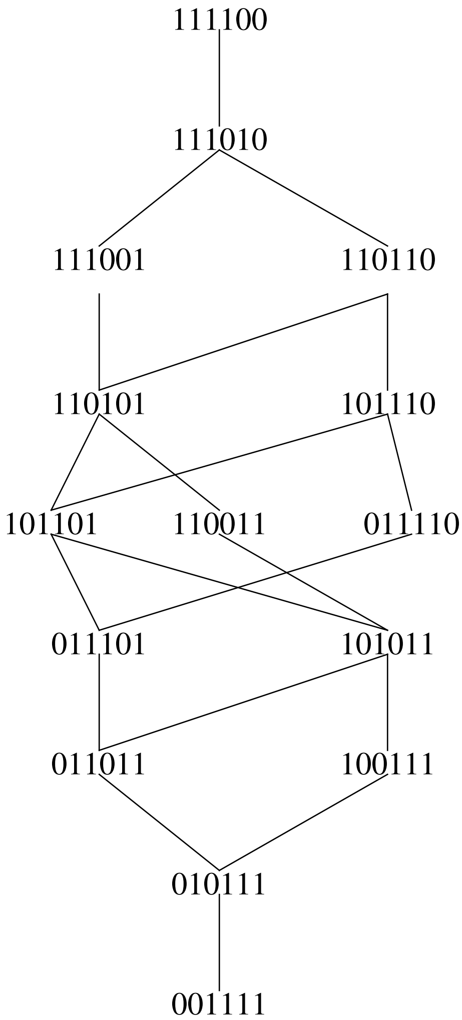
Satz

Es gibt genau dann einen adjacent revolving door Durchlauf für die k Teilmengen einer n elementigen Grundmenge wenn gilt:

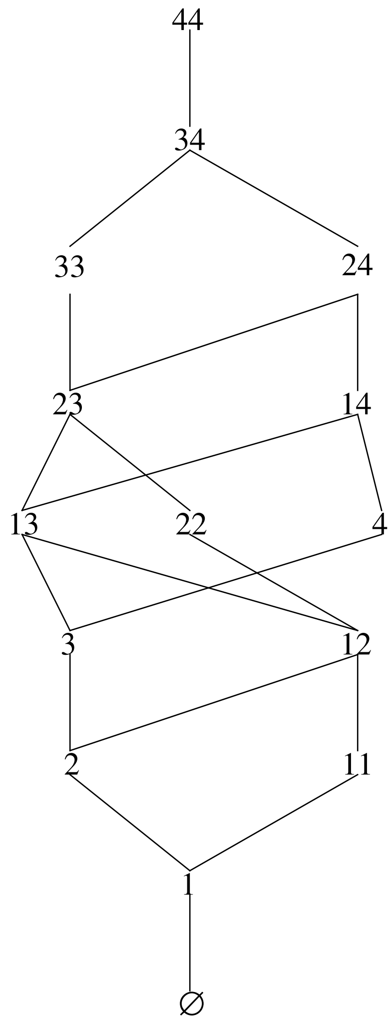
1. k aus {0,1,n-1,n} oder
2. k ist ungerade und n ist gerade

k-Teilmengen Poset

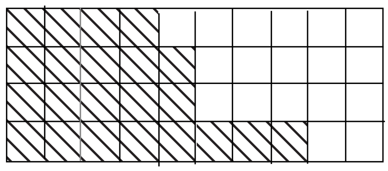
Man kann auch die k-Teilmengen in einem Poset anordnen, benachbart sind zwei Teilmengen wenn sie durch Verschieben einer 1 in der Binärdarstellung zu einer benachbarten 0 entstehen, d.h. beim revolving door Algorithmus wird ein Element durch seinen Vorgänger oder Nachfolger ersetzt, dies ist die adjacent revolving door relation. Als Beispiel folgt das Poset *S(6,4)* der vierelementigen Teilmengen einer sechselementigen Grundmenge. Dies Poset ist symmetrisch, die Symmetrie entsteht durch Lesen der Binärdarstellung von rechts bzw von links. Das Poset hat ein Minimum und ein Maximum. Das Poset ist ein Verband. Das Poset hat eine rank-Funktion.



Dies Poset kann man als ein Poset von Partitionen betrachten wenn man die 0-1 Folgen als Binärkodierung [Seite 46] von Partitionen liest, dann sieht dieses Poset so aus:



Es ist sofort klar was das ranking ist, das Gewicht der Partition, und die Regel wann zwei k-Teilmengen benachbart sind, ist das Entfernen eines 1-Haken, d.h. zwei Teilmengen sind benachbart wenn sie im Youngverband benachbart sind. Man sieht, daß im Poset *S(n,k)* die Partitionen enthalten sind, dessen Ferrers Diagramm in ein Rechteck mit k Spalten und n-k Zeilen paßt. Man somit gesehen, daß der Verband *S(n,k)* ein Teilverband des Youngverbands $\mathcal{YP}$ ist. Die Symmetrie des Verbandes sieht man indem man den Teil des Rechtecks nimmt, der nicht vom Ferrers Diagramm benötigt wird.



Im Beispiel sind die Partitionen (4,5,5,8) und (2,5,5,6) unter dieser Bijektion verbunden.

rank und unrank zum revolving door Algorithmus

Aufgaben

Partitionen

Übungsaufgabe:
Sei (m,h) = die Anzahl der Möglichkeiten die Zahl h als Summe von m+1 positven Zahlen zu schreiben .Z.B.(2,3)= 10=(003,012,021,030,102,111,120,201,210,300).

Zeige, daß (m,h)=:C(m,h)=$\binom{m+h}{h}$

Beweis:
Es gilt für die Binomialkoeffizienten die Rekursion:

$$\binom{m+h}{h} = \binom{(m-1)+h}{h} + \binom{m+(h-1)}{(h-1)}$$

Die Anzahl der Möglichkeiten die h als Summe von m+1 positven Zahlen zu schreiben spaltet man entsprechend der letzten (m+1-ten) Zahl auf, sie kann 0 oder größer sein. Die Anzahl mit 0 ist gleich der Anzahl der Möglichkeiten die Zah h als Summe von positven Zahlen zu schreiben. Ist der letzte Eintrag > 0, so kann man 1 an der letzten Stelle abziehen und erhält so, daß die Anzahl der Möglichkeiten = der Anzahl der Möglichkeiten die Zahl h-1 als Summe von m+1 positven Zahlen zu schreiben ist. D.h man hat die gewünschte Rekursion:

$$(m,h) = (m-1,h) + (m,h-1)$$

Man zeigt noch, daß die Rekursionsbasis gleich ist:

$$(m,0) = (0,h) = \binom{m+0}{0} = \binom{0+h}{h} = 1$$

Womit die Behauptung bewiesen ist.

Obige Übungsaufgabe zeigt daß die Anzahl der uneigentlichen Partitionen von h in m+1 Teilen gleich ist der Anzahl der uneigentlichen Partitionen von m in h+1 Teilen.

Übungsaufgabe:
Finde eine entsprechende Bijektion

Beispiel: Betrachte (2,10) = Partitionen von 10 in 3 Teilen ist in Bijektion zu (10,2) = Partitionen von 2 in 11 Teilen. Die Partition von 10 in 3 Teilen wird durch Auswahl von 2 Elementen aus 12 Elementen kodiert.

			X						X		

Die Anzahl der nicht gewählten Elemente bis zum gewählten gibt den Teil der Partition, obiges Beispiel entspricht der uneigentlichen Partition 3,5,2 von 10 in 3 Teilen . Nun gibt es eine natürliche Bijektion indem nichtgewählt und gewählt die Rolle vertauschen, so erhält man die uneigentliche Partition 0,0,0,1,0,0,0,0,1,0,0 von 2 in 11 Teilen. Dies ergibt die gesuchte Bijektion. Man kann natürlich mit dieser Kodierung auch obige Formel beweisen.

Übungsaufgabe:
Was ist die erzeugende Funktion für die Partitionen in verschiedenen Teilen?

Antwort:
$(1+x)(1+x^2)(1+x^3)...$

Anzahl
bis 100
1 1 2 2 3 4 5 6 8 10 12 15 18 22 27 32 38 46 54 64 76 89 104 122 142 165 192 222 256 296 340 390 448 512 585 668 760 864 982 1113 1260 1426 1610 1816 2048 2304 2590 2910 3264 3658 4097 4582 5120 5718 6378 7108 7917 8808 9792 10880 12076 13394 14848 16444 18200 20132 22250 24576 27130 29927 32992 36352 40026 44046 48446 53250 58499 64234 70488 77312 84756 92864 101698 111322 121792 133184 145578 159046 173682 189586 206848 225585 245920 267968 291874 317788 345856 376256 409174 444793

Übungsaufgabe:
Finde eine unendliche Famile von Partitionen ohne k-Haken.

Antwort:
(1,1+(k-1)), (1, 1+(k-1), 1 + 2*(k-1)), ...
allgemein
(1,...,1+i * (k-1), ...)
man sieht dies am einfachsten am 0-1 Rahmen, die 0 stehen dann genau im Abstand k zueinander. Andere Familien erhält man indem man nicht nur eine Menge von 0‘en im Abstand k betrachtet sondern mehrere.

Übungsaufgabe:
Finde eine Bijektion zwischen den Partitionen von n in verschiedenen Teilen und den Partitionen von n mit ungeraden Teilen.

Antwort: [Stanton White p.70]
Sei λ eine Partition mit ungeraden Teilen, sei der Teil i genau m_i mal vorhanden. Schreibe m_i in Binärschreibeweise $m_i = \Sigma\, a_k 2^k$ Ersetze nun die m_i durch die Teile $(i\cdot a_k 2^k)$. So wird 3^3 ersetzt durch die beiden Teile 3 und 6. Es ist klar daß die neuen Teile die zu m_i gehören alle verschieden sind, und es ist klar, daß sie auch verschieden sind wenn sie zu verschiedenen i gehören.
Beispiel:

1 1 1 1 3 3 3 5 5 5 9 9 15

4   3 6   5 10   18   15

unter obiger Bijektion wird die Partition 1^4 3^3 5^3 9^2 15 mit ungeraden Teilen abgebildet auf die Partition 3,4,5,6,10,15,18 mit verschiedenen Teilen.

Tableaux

Übungsaufgabe:

Finde eine Bijektion zwischen den Tableaux vom Umriß λ und dem Inhalt $c_1,..,c_i,c_{i+1},...$ und den Tableaux vom Umriß λ und dem Inhalt $c_1,..,c_{i+1},c_i,...$

Antwort:

Kodiere ein Tableau als 0-1 Matrix in 1. Quadranten, eine 1 am Platz (i,j) wenn in der j-ten Spalte die Zahl i vorkommt, sonst 0'en. Betrachte nun die Zeile i und i+1, sie hat z.B. die Form

$$\begin{array}{cccccccc} 1 & 1 & 1 & \dots & & & 1 & 1 \\ & 1 & 1 & \dots & 1 & 1 & & \end{array}$$

es werden nun 1'en markiert, alle die übereinander stehen und anschließend Paare mit der 1 in der Zeile i+1 und einer 1 in der Zeile i weiter rechts. Übrig bleibt etwas der Form

$$\begin{array}{cccccccc} & & & \dots & & & 1 & 1 \\ 1 & 1 & 1 & \dots & 1 & 1 & & \end{array}$$

(vorausgesetzt man hat mehr i als i+1) Um das symmetrische Tableau zu finden werden nun rechte 1'en aus der Zeile i in die Zeile i+1 geschoben, so daß danach in der Zeile i+_1 so viel 1'en sind wie vorher in der Zeile i+1, im obigen Beispiel:

$$\begin{array}{cccccccc} & & 1 & \dots & 1 & 1 & 1 & 1 \\ 1 & 1 & & \dots & & & & \end{array}$$

Dies ist das Tableau unter der gesuchten Bijektion. Hat das Tableaux mehr i+1 als i, so wird die inverse Abbildung genommen.

Source Code

In den nachfolgenden Kapiteln werden konkrete Implementierungen von im Text vorgestellten Algorithmen und Datenstrukturen besprochen.

Definition des Datentyp _MY_PERM

source code	Kommentar

`/* permutations */`
`typedef`
`struct permutation {`
`        int laenge;`
`        int * werte;`
`} _MY_PERM;`
 	
`#define SPL(a)  (a)->laenge`
`#define SPW(a)  (a)->werte`
	
	Die Anzahl der Elemente
	
	Die Elemente
	
	Für den Zugriff

Hilfsroutinen

source code	Kommentar

`#include "_my.h"`
`_my_error(int i)`
`        {`
`        switch (i)`
`                {`
`        case _MY_NO_MEM:`
`        fprintf(stderr,_MY_NO_MEM_TEXT);`
`        break;`
 	
`        case _MY_GROESSER_EINS:`
`        fprintf(stderr,_MY_GROESSER_EINS_TEXT);`
`        break;`
 	
`        default:`
`        fprintf(stderr,"Die Fehlernummer`
`%d",i);`
`        fprintf(stderr,"ist nicht bekannt\n");`
`        break;`
`                }`
`        }`
	
	Fehlerbehandlung

`char * _my_calloc(int i, int s)`
`        {`
`        char *res;`
`        res = calloc(i,s);`
`        if (res == NULL)`
`                _my_error(_MY_NO_MEM);`
`        return res;`
`        }`
	Eigenes Interface zu calloc

`int _my_free(char *a)` `        {free(a);}`	**Eigenes Interface zu free**

Permutationen I/O

source code	Kommentar

`#include "_my.h"`
`_my_scan_permutation(_MY_PERM *a)`
`{`
`        int i,j;`
`again:`
`        scanf("%d",&i);`
`        if (i<1) {`
`                _my_error(_MY_GROESSER_EINS);`
`                goto again;`
`                }`
`        else    {`
`                SPL(a) = i;`
`                SPW(a) =`
`                     _my_calloc(i,sizeof(int));`
`                for (j=0;j<i;j++)`
`                        scanf("%d", SPW(a)+j);`
`                }`
`}`
	Einlesen einer Permutation von stdin

`_my_print_permutation(_MY_PERM *a)`
`{`
`        int i;`
`        printf("[");`
`        for (i=0;i<SPL(a);i++)`
`                {`
`                printf("%d",* (SPW(a)+i) );`
`                if (i+1 < SPL(a))`
`                        printf(",");`
`                }`
`        printf("]\n");`
`}`
	Ausgabe einer Permutation auf stdout

Löschen einer Permutation

source code	Kommentar

`#include "_my.h"`
`_my_free_permutation(_MY_PERM *a)`
`/* gibt den Zeiger mit den Eintraegen frei */`
`{`
`        _my_free(SPW(a));`
`}`
	Löschen der Werte in einer Permutation, nicht der Struktur selber

Beispielprogramm

source code	Eingabe	Ausgabe

`#include "_my.h"`
 	
`main()`
`{`
`_MY_PERM p;`
`_my_scan_permutation(&p);`
`_my_print_permutation(&p);`
`_my_free_permutation(&p);`
`}`
	4 2 3 4 1	**[2,3,4,1]**

Gruppenoperationen für Permutationen

source code	Kommentar

`#include "_my.h"`
`_my_mult_permutation(_MY_PERM *a, _MY_PERM *b, _MY_PERM *c)`
`{`
`int i;`
`_my_free_permutation(c);`
`if (SPL(a) != SPL(b))`
`return`
`_my_error(_MY_DIFF_PERMSIZE);`
`SPL(c) = SPL(a);`
`SPW(c) =`
`_my_calloc(SPL(c),sizeof(int));`
`for (i=0;i<SPL(c);i++)`
`SPW(c)[i] = SPW(a)[SPW(b)[i]-1];`
`}`
	Multiplikation zweier Permutationen von rechts, d.h. erst b dann a.
	
	Funktioniert nur bei gleichem Grad
	
	-1 da arrays in C beim Index 0 beginnen.

`_my_invers_permutation(_MY_PERM *a,_MY_PERM *c)`
`{`
`int i;`
`_my_free_permutation(c);`
`SPL(c) = SPL(a);`
`SPW(c) =`
`_my_calloc(SPL(c),sizeof(int));`
`for (i=0;i<SPL(c);i++)`
`SPW(c)[SPW(a)[i]-1] = i+1;`
`}`
	Invertieren einer Permutation, Ergebnis in c
	
	c freigeben
	
	-1 da arrays in C beim Index 0 beginnen.

Löschen einer Permutation (neu), Initialisieren

Es gibt ein Problem mit dem Löschen innerhalb von invers und mult, was ist wenn die Permutation noch leer ist, daher:

source code	Kommentar

`#include "_my.h"`
`_my_free_permutation(_MY_PERM *a)`
`/* gibt den Zeiger mit den Eintraegen frei */`
`{`
`        if (SPW(a) != NULL)`
`                _my_free(SPW(a));`
`}`
`_my_init_permutation(_MY_PERM *a)`
`{`
`        SPL(a) = 0;`
`        SPW(a) = NULL;`
`}`
	
	Löschen der Werte in einer Permutation, nicht der Struktur selber
	
	Initialisieren zu Anfang

Beispielprogramm

Die Permutationen müssen vor Gebrauch initialisiert werden.

source code	Eingabe	Ausgabe

`#include "_my.h"`
 	
`main()`
`{`
`_MY_PERM a,b,c,d;`
`_my_init_permutation(&a);`
`_my_init_permutation(&b);`
`_my_init_permutation(&c);`
`_my_init_permutation(&d);`
`_my_scan_permutation(&a);`
`_my_scan_permutation(&b);`
`_my_mult_permutation(&a,&b,&c);`
`_my_print_permutation(&c);`
`_my_invers_permutation(&a,&d);`
`_my_print_permutation(&d);`
`_my_free_permutation(&a);`
`_my_free_permutation(&b);`
`_my_free_permutation(&c);`
`_my_free_permutation(&d);`
`}`
	`4 2 3 4 1`
	`4 3 2 4 1`
		`[4,3,1,2]`
		`[4,1,2,3]`

Umwandlung in Zykelschreibweise

<table>
<tr><th>source code</th><th>Kommentar</th></tr>
<tr><td>

```
#include "_my.h"
_my_list2zykel_permutation(_MY_PERM *a,
                           _MY_PERM *c)
{
        int i,j,m,k,n;
        _my_free_permutation(c);
        SPL(c) = SPL(a);
        SPW(c) =
              _my_calloc(SPL(c),sizeof(int));
        i = 0; k = 0; m = 1;
        n = SPL(c);
again:
        SPW(c)[k]=i+1;
        i = SPW(a)[i]-1;
        k++;
        if (i+1 != m) goto again;
        /* zykel zu ende */
        /* zykel der laenge k */
        /* nach rechts schieben */
        n = n - k;
        if (n == 0) return;
        memmove(SPW(c)+n, SPW(c),
                k *sizeof(int));
        /* nun den naechst kleineren zykel
           suchen */
        m++;
nochmal:
        for (i=0;i<SPL(c);i++)
                if (SPW(c)[i] == m)
                        { m++; goto nochmal; }
        /* m ist der naechste zykelanfang */
        i = m-1; k = 0;
        goto again;
}
```

</td><td>

Umwandlung von Listen-schreibweise in Zykel-schreibweise

Die Länge des Vektors bleibt gleich

m ist das kleinste Element im aktuellen Zykel

k ist die Zykellaenge

n ist der noch freie Platz im Ergebnis
Der gefundene Zykel wird nach rechts geschoben

</td></tr>
</table>

Beispielprogramm

<table>
<tr><th>source code</th><th>Eingabe</th><th>Ausgabe</th></tr>
<tr><td>

```
#include "_my.h"
main()
{
_MY_PERM a,b;
_my_init_permutation(&a);
_my_init_permutation(&b);
_my_scan_permutation(&a);
_my_list2zykel_permutation(&a,&b);
_my_print_permutation(&b);
_my_free_permutation(&a);
_my_free_permutation(&b);
}
```

</td><td>

10
2 4 3 6 7 5 1 10 9 8

</td><td>

[9,8,10,3,1,2,4,6,5,7]

</td></tr>
</table>

Invertieren am Platz

source code	Kommentar

`#include "_my.h"`
`_my_inversapply_permutation(_MY_PERM *a)`
`/* nach knuth I,p.173 */`
`{`
`        int m,i,j,k;`
`        m = SPL(a);`
`        while (m>0) {`
`                /* I2 */`
`                i = SPW(a)[m-1];`
`                if (i == m+1) {m--;continue;}`
`                if (i < 0) {SPW(a)[m-1] = -i;`
`                            m--;continue;}`
`                k = m; /* I3 */`
`                I4: j = SPW(a)[i-1];`
`                    SPW(a)[i-1] = -k;`
`                if (j == m ) SPW(a)[m-1] = i;`
`                else {k=i;i=j;goto I4;}`
`                /* I6 */ m--;`
`        }`
`}`
	invertieren ohne Hilfs-feld
	vgl. Algorithmus im Skript

Beispielprogramm

source code	Eingabe	Ausgabe

`#include "_my.h"`
`main()`
`{`
`_MY_PERM a,b;`
`_my_init_permutation(&a);`
`_my_scan_permutation(&a);`
`_my_print_permutation(&a);`
`_my_inversapply_permutation(&a);`
`_my_print_permutation(&a);`
`_my_free_permutation(&a);`
`}`
	`5 2 5 4 3 1`	`[2,5,4,3,1]`
		`[5,1,4,3,2]`

Lehmercode

<table>
<tr><th>source code</th><th>Kommentar</th></tr>
<tr><td><pre>
#include “_my.h“
_my_lehmercode_permutation(_MY_PERM *a,
                           _MY_PERM *b)
{
        int i,j;
        _my_free_permutation(b);
        SPL(b) = SPL(a);
        SPW(b) =
                _my_calloc(SPL(b),sizeof(int));
        for (i=0;i<SPL(a);i++)
        for (j=i+1; j<SPL(a); j++)
                if (SPW(a)[j] < SPW(a)[i])
                                SPW(b)[i]++;
}
</pre></td><td><pre>

Lehmercode
vgl. Algorithmus im
Skript

Zählen rechts die klei-
ner sind
</pre></td></tr>
<tr><td><pre>
_my_invlehmercode_permutation(_MY_PERM *a,
                              _MY_PERM *b)
{
        int i,j,k;
        int *l;
        _my_free_permutation(b);
        SPL(b) = SPL(a);
        SPW(b) =
                _my_calloc(SPL(b),sizeof(int));
        l = _my_calloc(SPL(b),sizeof(int));
        for (i=0;i<SPL(b);i++)
                l[i]=i+1;
        for (i=0;i<SPL(b);i++)
                {
                k = 0; j = 0;
                while (k <= SPW(a)[i])
                        if (l[j++] > 0) k++;
                /* an der stelle j-1 steht der
eintrag fuer die perm. */
                SPW(b)[i] = l[j-1];
                l[j-1]=0;
                }
        _my_free(l);
}
</pre></td><td><pre>
inverse Abbildung

l ist das Hilfsfeld mit
den Werten 1...n

suchen des i-ten Wert
im Hilfsfeld

den gebrauchten Wert
auf 0 setzen
</pre></td></tr>
</table>

Beispielprogramm

source code	Eingabe	Ausgabe

`#include “_my.h“`
`main()`
`{`
`_MY_PERM a,b;`
`_my_init_permutation(&a);`
`_my_init_permutation(&b);`
`_my_scan_permutation(&a);`
`_my_print_permutation(&a);`
`_my_lehmercode_permutation(&a,&b);`
`_my_print_permutation(&b);`
`_my_invlehmercode_permutation(&b,&a);`
`_my_print_permutation(&a);`
`_my_free_permutation(&a);`
`_my_free_permutation(&b);`
`}`
	6
	3 2 4 6 5 1
		
		`[3,2,4,6,5,1]`
		
		`[2,1,1,2,1,0]`
		
		`[3,2,4,6,5,1]`

rank/unrank

<table>
<tr><th>source code</th><th>Kommentar</th></tr>
<tr><td><pre>
#include “_my.h“
_my_rank_permutation(_MY_PERM *a, int *r)
{
        _MY_PERM b;
        int i,j;
        _my_init_permutation(&b);
        _my_lehmercode_permutation(a,&b);
        for (i=0, *r=0;i<SPL(&b);i++)
                {
                _my_fakul(SPL(a)-1-i, &j);
                *r += SPW(&b)[i]*j;
                }
        _my_free_permutation(&b);
}
</pre></td><td>ranking mittels erster Basis = fakultät

Lehmercode = Koeffizienten für i!</td></tr>
<tr><td><pre>
_my_unrank_permutation(int r, int n,
                             _MY_PERM *a)
{
        int i,f;
        _MY_PERM b;
        _my_init_permutation(&b);
        _my_free_permutation(a);
        SPL(a) = n;
        SPW(a) = (int *)
                 _my_calloc(SPL(a),sizeof(int));
        SPL(&b) = n;
        SPW(&b) = (int *)
                 _my_calloc(SPL(a),sizeof(int));

        _my_fakul(n,&f);
        for (i=0;i<SPL(a);i++)
                {
                f /= (SPL(a)-i);
                SPW(&b)[i] = r / f;
                r %= f;
                }

        _my_invlehmercode_permutation(&b,a);
        _my_free_permutation(&b);
}
</pre></td><td>inverse Abbildung

Fakultät wird nur einmal berechnet

inverser Lehmercode</td></tr>
</table>

Beispielprogramm

source code	Eingabe	Ausgabe

`#include "_my.h"`
`main()`
`{`
`_MY_PERM a;`
`_MY_PERM b;`
`int i;`
`_my_init_permutation(&a);`
`_my_init_permutation(&b);`
`_my_scan_permutation(&a);`
`_my_print_permutation(&a);`
`_my_rank_permutation(&a,&i);`
`printf("rank = %d\n",i);`
`_my_unrank_permutation(i,SPL(&a),&b);`
`_my_print_permutation(&b);`
`_my_free_permutation(&a);`
`_my_free_permutation(&b);`
`}`
	7
	3 2 1 7 6 5 4
		`[3,2,1,7,6,5,4]`
		
		`rank = 1583`
		
		`[3,2,1,7,6,5,4]`

Test auf Permutation

source code	Kommentar
`#include "_my.h"`	
`_my_scan_permutation(_MY_PERM *a)`	
`{`	
`        int i,j;`	
`again:`	
`        .....`	bekannte routine
`          if (_my_deeptest_permutation(a))`	Fehler falls <> 0
`              {`	
`              _my_error(_MY_KEINE_PERMUTATION);`	
`               goto again;`	
`               }`	
`}`	
`_my_deeptest_permutation(_MY_PERM *a)`	
`{`	mögliche Fehler
`        int i, *c, res=0;`	NULL Zeiger
`        if (a == NULL)`	
`                return 1;`	
`        if (SPL(a) < 1)`	falscher Grad
`                return 1;`	
`        c = (int *)_my_calloc(SPL(a),`	
`                              sizeof(int));`	
`        for (i=0;i<SPL(a);)`	Hilfsfeld mit 1...n
`                c[i] = ++i;`	
`        for (i=0;i<SPL(a);)`	
`                {`	
`                if (`	
`                        (SPW(a)[i] < 1)  ||`	Wert zu klein
`                        (SPW(a)[i] > SPL(a) )`	Wert zu groß
`                   )    {`	
`                        res = 1;`	
`                        break;`	
`                        }`	
`                if (c[SPW(a)[i] -1] == 0)`	Wert schon mal vorge-kommen
`                        {`	
`                        res = 1;`	
`                        break;`	
`                        }`	
`                c[SPW(a)[i] -1] = 0;`	
`                }`	
`        _my_free(c);`	
`        return res;`	
`}`	

random

source code	Kommentar

`#include "_my.h"`
`_my_random_permutation(int grad, _MY_PERM *b)`
`{`
`        _MY_PERM c;`
`        int i;`
`        _my_init_permutation(&c);`
`        _my_free_permutation(b);`
`        SPL(&c) = grad;`
`        SPW(&c) = (int *)`
`                 _my_calloc(grad,sizeof(int));`
`        SPL(b) = grad;`
`        SPW(b) = (int *)`
`                 _my_calloc(grad,sizeof(int));`
 	
`        srand(getpid()); /* unix */`
`        for (i=grad-2;i>=0;i--)`
`                {`
`                SPW(&c)[i]=rand()%(grad-1-i);`
`                }`
 	
`        _my_invlehmercode_permutation(&c,b);`
`        _my_free_permutation(&c);`
`}`
	random mittels Lehmer-code
	
	Aufbau des Lehmercodes
	
	inverser Lehmercode

Beispielprogramm

source code	Eingabe	Ausgabe

`#include "_my.h"`
`main()`
`{`
`_MY_PERM a;`
`int i;`
`scanf("%d",&i);`
`_my_init_permutation(&a);`
`_my_random_permutation(i,&a);`
`_my_print_permutation(&a);`
`_my_free_permutation(&a);`
`}`
	
	12
		
		[4,1,2,11,3,7,10,9,5,8,6,12]

Zufällige Permutation von festen Grad und fester Anzahl von Inversionen

Anzahl (Hilfsroutine)

source code	Kommentar

<pre>#include "_my.h"
static in(int m,int n,int k, int *feld)
/* erzeugt ein feld mit (n,k) =
anzahl perm von n mit k inversionen */
{
        int i;
        int res = 0;
        if (k < 0) return 0;
        if (k == 0)
                {
                * (feld + (m*k) + n) = 1;
                return 1;
                }
        if (n == 1) return 0;
 	
        if (* (feld + (m*k) + n) != 0) return
                          * (feld + (m*k) + n);
 	
        for (i=0; i<n ; i++)
                res += in(m,n-1,k-i,feld);
        * (feld + (m*k) + n) = res;
        return res;
}</pre>
	
	Hilfsroutine zur Anzahl Berechnung mittels Feld zum Zwischenspeichern

Anzahl

source code	Kommentar

<pre>#include "_my.h"
 	
_my_number_permutation_in(int n, int k, int
*res)
{
int *feld = (int *) _my_calloc((n+1) * (k+1) ,
            sizeof(int));
*res = in(n+1,n,k, feld);
_my_free((signed char *)feld);
}</pre>
	
	Routine zur Anzahl Berechnung
	
	Hilfsfeld

Zufällige Permutation mit gegebener Zahl von Inversionen.

source code	Kommentar
`#include “_my.h“`	
	Routine zur Anzahl Berechnung
`_my_random_permutation_in(int n, int k,`	
`                          _MY_PERM *c)`	
`{`	
`int *feld = (int *) _my_calloc((n+1) * (k+1) ,`	Hilfsfeld
`                     sizeof(int));`	
`int p,sum,osum,j,res,gewicht;`	
`_MY_PERM b;`	
`_my_free_permutation(&b);`	
`SPL(&b) = n;`	
`SPW(&b) = (int *)_my_calloc(n, sizeof(int));`	Platz für Lehmercode
`        /* hilfsfeld fuer lehmer code */`	
`res = in(n+1,n,k, feld);`	Anzahl
`srand(getpid());`	passende Zufallszahl
`p = rand() %  res;`	
`gewicht = n-1;`	
`again:`	Abarbeiten der Rekursion
`        for (j = sum = osum = 0; ; j++)`	
`        {`	
`        osum = sum;`	
`        sum += * (feld + ((n+1)*(k-j)) +`	
`                                   gewicht);`	
`        if (sum >= p) break;`	
`        }`	
`SPW(&b)[n - 1  - gewicht] = j;`	
`p = p - osum;`	
`k = k - j;`	
`gewicht--;`	
`if (gewicht > 0) goto again;`	
`if (k != 0)`	
`        _my_error(_MY_INTERNAL_ERROR);`	
`_my_invlehmercode_permutation(&b,c);`	Lehmercode in Permutation umwandeln
`_my_free_permutation(&b);`	
`_my_free((signed char *)feld);`	
`}`	

Beispielprogramm

source code	Eingabe	Ausgabe

`#include "_my.h"`
`main()`
`{`
`int n,k;`
`_MY_PERM b;`
`_my_init_permutation(&b);`
`scanf("%d",&n);`
`scanf("%d",&k);`
`_my_random_permutation_in(n,k,&b);`
`_my_print_permutation(&b);`
`_my_free_permutation(&b);`
 	
`}`
	
	15
	5
		
		[1,2,4,6,3,5,7,8
		,9,10,11,12,14,1
		5,13]

Durchlaufen der Permutationen mit gegebener Anzahl von Inversionen

erste Permutation.

source code	Kommentar
(see code below)	

```
#include "_my.h"


_my_first_permutation_in(int n, int k, _MY_PERM
*c)
{
        _MY_PERM b;
        int i;
        SPL(&b) = n;
        SPW(&b) = (int *)_my_calloc(n,
              sizeof(int));
               /* hilfsfeld fuer lehmer code */
        for (i=0;i<n;i++)
                {
                SPW(&b)[i] =
                      (k > n-1-i ? n-i-1 : k);
                k = k - (n-1-i);
                if ( k <= 0 ) break;
                }
        if (k > 0)
                _my_error(_MY_INTERNAL_ERROR);
        _my_invlehmercode_permutation(&b,c);
        _my_free_permutation(&b);
}
```

nächste Permutation.

<table>
<tr><th>source code</th><th>Kommentar</th></tr>
<tr><td><pre>
#include “_my.h“

_my_next_permutation_in(_MY_PERM *c, _MY_PERM
*b)
{
        int i,sum,j,res,k;
        _MY_PERM a;
        _my_init_permutation(&a);
        _my_lehmercode_permutation(c,&a);
        for (i=SPL(&a)-1,sum=0,j=0;i>=0;i--
                    ,j++)
                {
                sum += SPW(&a)[i];
                if (SPW(&a)[i] < j) break;
                }
        if (i < 0) { res = 1; goto ende; }
        for (j = i-1; j >= 0; j--)
                if (SPW(&a)[j] > 0) break;
        if (j < 0) { res = 1; goto ende; }
        res = 0;
        SPW(&a)[j] --;
        sum++;
        for (i = j+1, k = SPL(&a) - 1 - i; i <
                              SPL(&a); i++,k--)
                if (sum >= k)
                      { SPW(&a)[i]=k;sum -=k; }
                else
                      {SPW(&a)[i]=sum;sum= 0; }
        _my_invlehmercode_permutation(&a,b);
ende:
        _my_free_permutation(&a);
        return res;
}
</pre></td><td><pre>

Algorithmus mit Lehmer-
code

am Platz j erniedrigen,
rechts auffüllen
</pre></td></tr>
</table>

Beispiel.

<table>
<tr><th>source code</th><th>Eingabe</th><th>Ausgabe</th></tr>
<tr><td><pre>/* _p14.c */
#include "_my.h"

main()
{
int n,k;
_MY_PERM b;
_my_init_permutation(&b);
scanf("%d",&n);
scanf("%d",&k);

_my_first_permutation_in(n,k,&b);
do {
_my_print_permutation(&b);
} while (_my_next_permutation_in(&b,
                   &b) == 0);

_my_free_permutation(&b);}</pre></td><td><pre>

7
2</pre></td><td><pre>

[3,1,2,4,5,6,7]
[2,3,1,4,5,6,7]
[2,1,4,3,5,6,7]
[2,1,3,5,4,6,7]
[2,1,3,4,6,5,7]
[2,1,3,4,5,7,6]
[1,4,2,3,5,6,7]
[1,3,4,2,5,6,7]
[1,3,2,5,4,6,7]
[1,3,2,4,6,5,7]
[1,3,2,4,5,7,6]
[1,2,5,3,4,6,7]
[1,2,4,5,3,6,7]
[1,2,4,3,6,5,7]
[1,2,4,3,5,7,6]
[1,2,3,6,4,5,7]
[1,2,3,5,6,4,7]
[1,2,3,5,4,7,6]
[1,2,3,4,7,5,6]
[1,2,3,4,6,7,5]</pre></td></tr>
</table>

Berechnung der Anzahl der Partitionen

source code	Kommentar

`main()`
`{`
`int i;`
`scanf("%d",&i);`
`printf("Anzahl Partitionen zu %d = %d\n"`
`                              ,i,nm(i+1,1));`
`}`
 	
`nm(int n,int m)`
`{`
`        int i,res;`
`        if (n < m) return 0;`
`        if (n == m) return 1;`
`        res = 0;`
`        for (i=m; i<= n-m; i++)`
`                res += nm(n-m,i);`
`return res;`
`}`
	Einfache Implementierung der Rekursion

Laufzeiten

n	Anzahl	sec
10	42	0
20	627	0
30	5604	0
50	204226	0.1
70	4087968	1.9
90	56634173	29.6
100	190569292	103.7

Daher sollte das Programm so geändert werden, daß die Ergebnisse in nm zwischengespeichert werden. Die Implementierung könnte so aussehen:

source code	Kommentar

```
int *feld;
main()
{
int i;
scanf(“%d“,&i);
feld = (int *)calloc((i+1)*(i+1), sizeof(int));
printf(“Anzahl Partitionen zu %d = %d\n“,
                                 i,nm(i+1,i+1,1));
free(feld);
}

nm(int k,int n,int m)
{
        int i,res;
        if (n < m) return 0;
        if (n == m) return 1;
        if (*(feld + (k*n)+m) != 0)
                return *(feld + (k*n)+m);
        res = 0;
        for (i=m; i<= n-m; i++)
                res += nm(k,n-m,i);
        *(feld + (k*n)+m) = res;
        return res;
}
```

Implementierung der Rekursion mit Speichern der berechneten Ergebnisse.

Laufzeiten

n	Anzahl	sec
10	42	0
20	627	0
30	5604	0
50	204226	0
70	4087968	0
90	56634173	0
100	190569292	0

und das Problem jetzt ist die Langzahlarithmetik

Definition des Datentyp _MY_PART

source code	Kommentar

`/* partitions */`
`typedef`
`struct partition {`
`        unsigned int art:2;`
`        int laenge;`
`        int * werte;`
`} _MY_PART;`
 	
`#define SPL(a) (a)->laenge`
`#define SPW(a) (a)->werte`
`#define SPA(a) (a)->art`
`#define LISTE 0`
`#define FROBENIUS 1`
`#define EXPONENT 2`
	
	Welche Kodierung?
	Die Anzahl der Elemente
	Die Elemente
	
	Für den Zugriff

Die Standardroutine sind in Anlehnung an die Implementierung der Permutationen.

Partitionen Einlesen

<table>
<tr><th>source code</th><th>Kommentar</th></tr>
<tr><td>

```
#include "_my.h"
_my_scan_partition(_MY_PART *a)
{
        int i,j;
again:  scanf(„%d",&i);
        if (i<1) {
                _my_error(_MY_GROESSER_EINS);
                goto again;
                }
        else    {
                SPL(a) = i;
                SPW(a) = (int
                  *)_my_calloc(i,sizeof(int));
                for (j=0;j<i;j++)
                        scanf("%d", SPW(a)+j);
                }
        if (_my_deeptest_partition(a))
                {
                _my_error(_MY_KEINE_PARTITION);
                goto again;
                }
        SPA(a) = LISTE;
}
```

</td><td>

Einlesen einer Partition von stdin

Test der Eingabe

</td></tr>
</table>

Ausgabe einer Partition

<table>
<tr><th>source code</th><th>Kommentar</th></tr>
<tr><td>

```
#include "_my.h"
_my_print_partition(_MY_PART *a)
{
        int i;
        if (SPA(a) == FROBENIUS)
                {
                }
        else    {
                printf("[");
                for (i=0;i<SPL(a);i++)
                        {
                    printf("%d",*(SPW(a)+i) );
                        if (i+1 < SPL(a))
                                printf(",");
                        }
                printf("]\n");
                }
}
```

</td><td>

funktioniert nur bei LISTE und EXPONENT Typ

</td></tr>
</table>

Testen einer Partition

source code	Kommentar
`#include "_my.h"`	
`_my_deeptest_partition(_MY_PART *a)`	`funktioniert nur bei`
`{`	`LISTE Typ`
`        int i, *c, res=0;`	
`        if (a == NULL)`	
`                return 1;`	
`        if (SPL(a) < 1)`	
`                return 1;`	
`        if (SPW(a)[0] <= 0)`	
`                        return 1;`	
`        for (i=1;i<SPL(a);i++)`	
`                {`	
`                if (SPW(a)[i] < SPW(a)[i-1])`	
`                        return 1;`	`Test ob ansteigend`
`                }`	
`        return 0;`	
`}`	

Initialisieren/Löschen

source code	Kommentar
`#include "_my.h"`	
`_my_free_partition(_MY_PART *a)`	
`/* gibt den Zeiger mit den Eintraegen frei */`	`Löschen`
`{`	
`        if (SPW(a) != NULL)`	
`                _my_free(SPW(a));`	
`}`	
`_my_init_partition(_MY_PART *a)`	`Initialisieren`
`{`	
`        SPL(a) = 0;`	
`        SPA(a) = LISTE;`	
`        SPW(a) = NULL;`	
`}`	

Beispiel

source code	Eingabe	Ausgabe

`#include "_my.h"`
 	
`main()`
`{`
`_MY_PART p;`
`_my_init_partition(&p);`
`_my_scan_partition(&p);`
`_my_print_partition(&p);`
`_my_free_partition(&p);`
`}`
	`4 1 3 5`	`[1,3,5]`

Umwandlung Liste/Exponent

source code	Kommentar

`#include "_my.h"`
 	
`_my_liste2exponent_partition(_MY_PART *a,`
`_MY_PART *b)`
`{`
`        int i;`
`        _my_free_partition(b);`
`        SPL(b) = SPW(a)[SPL(a)-1];`
`        SPW(b) = _my_calloc(SPL(b),`
`                 sizeof(int));`
`        SPA(b) = EXPONENT;`
`        for (i=0;i<SPL(a);i++)`
`                SPW(b)[SPW(a)[i] -1]++;`
`}`
	
	`Länge des Exponentenv.`
	`0 vorbesetzen`
	
	`Werte in Exp. erzeugen`

Beispiel

source code	Eingabe	Ausgabe

`#include "_my.h"`
 	
`main()`
`{`
`_MY_PART p,q;`
`_my_init_partition(&p);`
`_my_init_partition(&q);`
`_my_scan_partition(&p);`
`_my_print_partition(&p);`
`_my_liste2exponent_partition(&p,&q);`
`_my_print_partition(&q);`
`_my_free_partition(&p);`
`_my_free_partition(&q);`
`}`
	
	`4 1 3 5`
		
		`[1,3,5]`
		
		`[1,0,1,0,1]`

Kopieren

source code	Kommentar
`#include “_my.h“`	
`_my_copy_partition(_MY_PART *a, _MY_PART *b)`	
`{`	
`if (SPL(b) != SPL(a))`	`nur neu erzeugen wenn`
`{`	`noch nicht vorhanden`
`_my_free_partition(b);`	
`SPL(b) = SPL(a);`	
`SPW(b) = (int *)`	
`_my_calloc(SPL(a),sizeof(int));`	`Vektor mit 0 vorbesetzt`
`}`	
`memcpy(SPW(b), SPW(a), SPL(a) *`	
`sizeof(int));`	`Werte kopieren`
`SPA(b) = SPA(a);`	`Art kopieren`
`}`	

Gewicht

source code	Kommentar
`#include “_my.h“`	
`_my_gewicht_partition(_MY_PART *a)`	
`{`	
`int i,j;`	`Vorerst nur in Listen-`
`if (SPA(a) == LISTE)`	`schreibweise`
`{`	
`for (i=0,j=0;i<SPL(a);i++)`	
`j += SPW(a)[i];`	
`return j;`	
`}`	
`else`	
`return -1;`	
`}`	

nächste Partition

source code	Kommentar

`#include "_my.h"`
`_my_next_partition(_MY_PART *a, _MY_PART *b)`
`{`
`        int i,sum;`
`        if (SPA(a) != EXPONENT)`
`                return 1;`
`        if (SPL(a) == 1)`
`                return 1; /* kein next */`
 	
`        for (i=1; i<SPL(a); i++)`
`                if (SPW(a)[i] > 0) break;`
`        if (i == SPL(a))`
`                return 1; /* kein next */`
`        if (a != b)`
`               _my_copy_partition(a,b);`
`        if ((i == SPL(a) -1)`
`             && (SPW(a)[SPL(a) -1] == 1) )`
`                SPL(b)--;`
`        sum  = SPW(a)[0]+i+1;`
`        SPW(b)[0]=0;`
`        SPW(b)[i]--;`
`        SPW(b)[i-1] = sum / i;`
`        SPW(b)[sum % i -1] = 1;`
`        return 0;`
`}`
	nur bei Exponententyp
	
	schon die letzte Partition
	
	es gab nach dem 1. Eintrag nur Nullen
	
	in diesem Fall wird der größte Teil kleiner
	
	Gewicht des Rests
	keine Einsen

erste Partition

source code	Kommentar

`#include "_my.h"`
 	
`_my_first_partition(int gewicht, _MY_PART *b)`
`{`
`        _my_free_partition(b);`
`        SPL(b) = gewicht;`
`        SPA(b) = EXPONENT;`
`        SPW(b) = (int *)`
`              _my_calloc(gewicht,sizeof(int));`
`        SPW(b)[gewicht-1]=1;`
`}`
	nur bei Exponententyp

Beispiel

source code	Eingabe	Ausgabe

`/* _pa16.c */`
`#include “_my.h“`
 	
`main()`
`{`
`_MY_PART p;`
`int i;`
`_my_init_partition(&p);`
`scanf(“%d“,&i);`
`_my_first_partition(i,&p);`
`do`
`        _my_print_partition(&p);`
`while ( _my_next_partition(&p, &p)`
`      == 0)  ;`
 	
`_my_free_partition(&p);`
`}`
	7	[0,0,0,0,0,0,1]
		[1,0,0,0,0,1]
		[0,1,0,0,1]
		[2,0,0,0,1]
		[0,0,1,1]
		[1,1,0,1]
		[3,0,0,1]
		[1,0,2]
		[0,2,1]
		[2,1,1]
		[4,0,1]
		[1,3]
		[3,2]
		[5,1]
		[7]

Bitvektor zur Speicherung von Partitionen

Dazu wird der Datentyp erweitert:

source code	Kommentar

<pre>/* partitions */
typedef
struct partition {
        unsigned int art:2;
        int laenge;
        int * werte;
} _MY_PART;
 	
...
#define BITVEKTOR 3</pre>	

Im Falle des Bitvektors speichert die Länge die Anzahl der Bits.

Routine zum Zugriff auf die Bits

source code	Kommentar

<pre>#include "_my.h"
_my_set_partition_bitvektor_i
                   (_MY_PART *b,int k,int v)
{
        char *zeiger =
                   (char *) SPW(b) + (k>>3);
        if (v == 0)
                *zeiger &= ~(1 << (k&7));
        else
                *zeiger |= (1 << (k&7));
 	
}
 	
_my_get_partition_bitvektor_i
                        (_MY_PART *b,int k)
{
        char *zeiger =
                   (char *) SPW(b) + (k>>3);
        return (*zeiger >> (k&7)) & 1;
}</pre>
	Setzen des k-ten Bits
	
	Lesen des k-ten Bits

Will man diese Routinen optimieren, so werden diese beiden Routinen Macros. Man beachte, daß die Operation &7 schneller ist als modulo 8, zumindest wenn man keinen guten Optimierer hat. Gleiches gilt für >>3 anstelle der Division durch 8.

Umwandlung von Listenkodierung zur Bitvektorkodierung

source code	Kommentar

`#include "_my.h"`
`_my_liste2bitvektor_partition`
`(_MY_PART *a, _MY_PART *b)`
`{`
`int i,j,k;`
`_my_free_partition(b);`
`SPL(b) = SPW(a)[SPL(a)-1]+SPL(a);`
`SPW(b) = (int *) _my_calloc(SPL(b)/8+1,`
`sizeof(char));`
`SPA(b) = BITVEKTOR;`
`i = 1;j = 0;k=0;`
`while (j < SPL(a))`
`{`
`while (i <= SPW(a)[j])`
`{`
`i++;`
`_my_set_partition_bitvektor_i`
`(b,k++,1);`
`}`
`_my_set_partition_bitvektor_i(b,k++,0);`
`j++;`
`}`
`while (k < (SPL(b)/8+1) * 8)`
`_my_set_partition_bitvektor_i`
`(b,k++,1);`
`}`
	Die Länge ist die Anzahl Bits
	
	Man wandert am Rahmen entlang, dabei ist i die Spalte und j die Zeile. k ist der Index im Bitvektor
	
	Auffüllen mit 1'en

Ausgabe des Bitvektors

Dazu wird die print Routine erweitert

source code	Kommentar

`#include "_my.h"`
`_my_print_partition(_MY_PART *a)`
`{`
`int i;`
`if (SPA(a) == FROBENIUS){}`
`else if (SPA(a) == BITVEKTOR)`
`{`
`printf("[");`
`for (i=0;i<SPL(a);i++)`
`printf("%d",`
`_my_get_partition_bitvektor_i(a,i));`
`printf("]\n");`
`}`
`else {`
`.....`
`}`
`}`
	Als Ausgabe 0-1 Folge innerhalb von [ ]

Beispiel

source code	Eingabe	Ausgabe

`/* _pa17.c */`
`#include "_my.h"`
 	
`main()`
`{`
`_MY_PART p,q;`
`_my_init_partition(&p);`
`_my_init_partition(&q);`
`_my_scan_partition(&p);`
`_my_print_partition(&p);`
`_my_liste2bitvektor_partition(&p,&q);`
`_my_print_partition(&q);`
`_my_free_partition(&p);`
`_my_free_partition(&q);`
`}`
	
	`4`
	`2 4 4 7`
		
		`[2,4,4,7]`
		
		`[11011001110]`

Berechnung der Permutationen mit gegebener Anzahl von Inversionen

source code	Kommentar

<pre>main()
{
int n,k;
scanf("%d",&n);
scanf("%d",&k);
printf("Anzahl Permutationen von %d mit %d
        Inversionen = %d\n",n,k,in(n,k));
}
 	
in(int n,int k)
{
        int i;
        int res = 0;
        if (k < 0) return 0;
        if (k == 0) return 1;
        if (n == 1) return 0;
 	
        for (i=0; i<n ; i++)
                res += in(n-1,k-i);
        return res;
 	
}</pre>
	Einfache Implementierung der Rekursion

Laufzeiten

n,k	Anzahl	sec
10,10	21670	0
15,10	836604	0.3
20,10	10947079	3.8
25,10	82164420	29.9

Daher sollte das Programm so geändert werden, daß die Ergebnisse in in zwischengespeichert werden. (Siehe Anzahl Partitionen). Danach dauert die Anzahlberechnung wieder 0.0 sec.

Will man die Anzahl der Aufrufe in der Rekursion zählen so kann man folgende Implementierung verwenden:

source code	Kommentar

<pre>int *feld;
 	
main()
{
int n,k,i,j;
scanf("%d",&n);
scanf("%d",&k);
feld = (int *) calloc((n+1) * (k+1) , sizeof(int));
printf("Anzahl Permutationen von %d mit %d Inversionen =
        %d\n",n,k,in(n+1,n,k));
for (i=0;i<=n;i++)
         {
         for (j=0;j<=k;j++)
                 printf("\t%d", * (feld + ((n+1)*j) + i));
         printf("\n");
         }
}
 	
in(int m,int n,int k)
{
         int i;
         int res = 0;
         if (k < 0) return 0;
         if (n <= 0) return 0;
         (*(feld + (m*k) + n))++;
         if (k==0) return 1;
 	
         for (i=0; i<n ; i++)
                 res += in(m,n-1,k-i);
         return res;
}</pre>
	Implementierung der Rekursion mit Speichern der Anzahl der Aufrufe.

Beispiel

Eingabe: 10 10

Ausgabe:Anzahl Permutationen von 10 mit 10 Inversionen = 21670

0	0	0	0	0	0	0	0	0	0	0
8483	13640	8095	4489	2298	1068	440	155	44	9	1
6887	8483	5157	2938	1551	747	321	119	36	8	1
3759	4214	2673	1596	888	454	209	84	28	7	1
1666	1792	1203	764	455	251	126	56	21	6	1
624	659	474	324	209	126	70	35	15	5	1
195	205	160	119	84	56	35	20	10	4	1
48	51	44	36	28	21	15	10	6	3	1
8	9	9	8	7	6	5	4	3	2	1
0	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	1

Definition des Datentyp _MY_TAB

source code	Kommentar
`/* tableaux */`	
`typedef`	
`        struct {`	
`        unsigned int art:2;`	`Welcher Typ?`
`        void *umriss;`	
`        void *werte;`	`Die Elemente`
`        }`	
`_MY_TAB;`	
`#define STA(a) (a)->art`	`Für den Zugriff`
`#define STU(a) ((_MY_PART *)(a)->umriss)`	
`#define STW(a) (a)->werte`	
`#define STZ(a) SPL(STU(a))`	`Zeilen`
`#define STS(a) SPW(STU(a))[STZ(a)-1]`	`Spalten`
`#define STIJ(a,i,j) *((int*)STW(a)+((i)*STS(a))+ j)`	`Einträge`
`#define INT_STANDARD_YOUNG_TAB 0`	
`/* Standard Young Tableau */`	
`/* umriss == partition vom typ LISTE */`	
`/* werte == int > 0 */`	

Die Standardroutine sind in Anlehnung an die Implementierung der Permutationen.

Tableau Einlesen

source code	Kommentar
`#include "_my.h"`	
`_my_scan_tab(_MY_TAB *a)`	**Einlesen eines Tableau**
`{`	**Erst der Umriß**
`int i,j;`	
`STU(a) = (void *)`	
`_my_malloc(sizeof(_MY_PART));`	
`_my_scan_partition(STU(a));`	
`STW(a) = _my_calloc(STZ(a) *`	
`STS(a),sizeof(int));`	
`for (i=STZ(a)-1; i>=0; i--)`	**Dann die Zeilen, zuerst**
`for (j=0;j<SPW(STU(a))[i]; j++)`	**die längste**
`scanf("%d", & STIJ(a,i,j));`	
`STA(a) = INT_STANDARD_YOUNG_TAB;`	
`}`	

Ausgabe eines Tableaus

source code	Kommentar
`#include "_my.h"`	
`_my_print_tab(_MY_TAB *a)`	
`{`	**Zeilenweise**
`int i,j;`	
`for (i=0; i<STZ(a); i++, printf("\n"))`	
`for (j=0;j<SPW(STU(a))[i]; j++)`	
`printf("%d ",STIJ(a,i,j));`	
`}}`	

Löschen

source code	Kommentar
`#include "_my.h"`	
`_my_free_tab(_MY_TAB *a)`	
`{`	**Löschen**
`_my_free_partition(STU(a));`	
`_my_free(STW(a));`	
`}`	

Beispiel

<table>
<tr><th>source code</th><th>Eingabe</th><th>Ausgabe</th></tr>
<tr><td><pre><code>#include “_my.h“
main()
{
        _MY_TAB p;
        _my_scan_tab(&p);
        _my_print_tab(&p);
        _my_free_tab(&p);
}</code></pre></td><td><pre><code>3 1 3 5
1 2 3 6 7
4 5 8
9 10</code></pre></td><td><pre><code>

9 10
4 5 8
1 2 3 6 7</code></pre></td></tr>
</table>

Zufälliges SYT

source code

```
#include "_my.h"

_my_random_tab(_MY_PART *p, _MY_TAB *a)
{
        int i,j,g,r,m,ii,jj,mr;
        STU(a) = (void *) _my_malloc(sizeof(_MY_PART));
        _my_copy_partition(p, STU(a));
        STW(a) = _my_calloc(STZ(a) * STS(a),sizeof(int));
        g = _my_gewicht_partition(p);
        while (g >  0)
                {
                r = rand() % g ;
                /* nun den platz finden */
                for (i=0,m= -1; i<STZ(a); i++)
                        for (j=0;j<SPW(STU(a))[i]; j++)
                                {
                                if (STIJ(a,i,j) > 0) continue;
                                else m++;
                                if (m == r) /* platz gefunden */
                                        {
                                        STIJ(a,i,j) = g;
                                        goto schieben;
                                        }
                                }
                _my_error(_MY_INTERNAL_ERROR);
schieben:       /* am platz i,j */
                m = 0;
                for (jj = 1; (j+jj < SPW(STU(a))[i]) &&
                         (STIJ(a,i,j+jj) == 0); jj++,m++);
                mr = m;
                for (ii = 1; ((i-ii) >=0 ) && (j < SPW(STU(a))[i-
                         ii]) && (STIJ(a,i-ii,j) == 0); ii++,m++);
                /* m ist die Anzahl der freien Plaetze im Haken */
                /* mr ist die Anzahl der freien Plaetze im rechten
                   Haken */
                if (m == 0)
                        {
                        g--;
                        }
                else    {
                        r = rand() % m ;
                        if (r >= mr) {r = r - mr;ii=i-r-1;jj= j;}
                        else { ii = i; jj = j + r+1; };
                        STIJ(a,i,j) = 0;
                        STIJ(a,ii,jj) = g;
                        i = ii; j = jj;
                        goto schieben;
                        };
                };
}
```

Beispiel

source code	Eingabe	Ausgabe

`#include "_my.h"`
 	
`main()`
`{`
`        _MY_TAB p;`
`        _MY_PART a;`
`        srand(getpid());`
`        _my_scan_partition(&a);`
`        _my_print_partition(&a);`
`        _my_random_tab(&a,&p);`
`        _my_print_tab(&p);`
`        _my_free_tab(&p);`
`        _my_free_partition(&a);`
`}`
	`3 1 3 5`	`[1,3,5]`
		`5`
		`2 7 8`
		`1 3 4 6 9`

Definition des Datentyp _MY_SUBSET

source code	Kommentar

`/* subset */`
`typedef`
`        struct {`
`        unsigned int art:2;`
`        int n; /*gesamtzahl*/`
`        int k; /* anzahl im subset */`
`        void *werte;`
`        }`
`_MY_SUBSET;`
`#define SSA(a)  (a)->art`
`#define SSN(a)  (a)->n`
`#define SSK(a)  (a)->k`
`#define SSW(a)  (a)->werte`
`#define SSWI(a,i)  *(((int*)((a)->werte)) + (i))`
`#define SS_LISTE 0`
`#define SS_BITVEKTOR 0`
	
	Welcher Typ?
	
	Die Elemente
	
	Für den Zugriff

Die Standardroutine sind in Anlehnung an die Implementierung der Permutationen.

Teilmenge Einlesen

source code	Kommentar

`#include "_my.h"`
`_my_scan_subset(_MY_SUBSET *a)`
`{`
`        int i,j;`
`again:`
`        scanf("%d", & SSN(a));`
`        scanf("%d", & SSK(a));`
`        SSW(a) =`
`              _my_calloc(SSK(a),sizeof(int));`
 	
`        for (j=0;j<SSK(a); j++)`
`                scanf("%d", & SSWI(a,j));`
`        SSA(a) = SS_LISTE;`
`}`
	
	Einlesen einer Teil-
	menge
	
	Erst die Gesamtanzahl
	Dann Anzahl der Teil-
	menge
	
	Einlesen der Elemente
	in der Teilmenge

Ausgabe einer Teilmenge

<table>
<tr><th>source code</th><th>Kommentar</th></tr>
<tr><td><pre>#include “_my.h“
static print_rd(int n, int k, int *sub)
{
        int i;
        for (i=0;i<k; i++)
        printf(“%d “,sub[i]);
        printf(“\n“);
}
_my_print_subset(_MY_SUBSET *a)
{
        return print_rd(SSN(a),SSK(a),SSW(a));
}</pre></td><td></td></tr>
</table>

Löschen

<table>
<tr><th>source code</th><th>Kommentar</th></tr>
<tr><td><pre>#include “_my.h“
_my_free_subset(_MY_TAB *a)
{
        _my_free(STW(a));
}</pre></td><td></td></tr>
</table>

Beispiel

<table>
<tr><th>source code</th><th>Eingabe</th><th>Ausgabe</th></tr>
<tr><td><pre>#include “_my.h“
main()
{
        _MY_SUBSET p;
        _my_scan_subset(&p);
        _my_print_subset(&p);
        _my_free_subset(&p);
}</pre></td><td><pre>5 2
2 5</pre></td><td><pre>2 5</pre></td></tr>
</table>

Nächste Teilmenge im Revolving Door Algorithmus

source code	Kommentar

`#include "_my.h"`
`static next_rd(int n, int k, int *s)`
`{`
`        int m,j;`
`        j = 0;`
`        if (k % 2 == 1) goto rd_n;`
`rd_v:`
`        j++;`
`        if (j>k) return 1;`
`        if (j == s[j-1]) goto rd_n;`
`        s[j-1]--;`
`        if (j > 1)  s[j-2]=j-1;`
`        return 0;`
`rd_n:`
`        j++;`
`        m = (j == k ? n : s[j] -1);`
`        if ( m == s[j-1]) goto rd_v;`
`        s[j-1]++;`
`        if (j > 1)  s[j-2]=s[j-1]-1;`
`        return 0;`
`}`
 	
`_my_next_k_subset(_MY_SUBSET *a)`
`{`
`        return next_rd(SSN(a), SSK(a), SSW(a));`
`}`
`static first_rd(int n, int k, int *sub)`
`{`
`        int i;`
`        for (i=0;i<k; i++)`
`        sub[i] = i + 1;`
`}`
`_my_first_subset(int n, int k, _MY_SUBSET *a)`
`{`
 	
`        SSW(a) = _my_calloc(k,sizeof(int));`
`        SSK(a) = k;`
`        SSN(a) = n;`
`        SSA(a) = SS_LISTE;`
`        return first_rd(SSN(a), SSK(a),`
`                                SSW(a));`
`}`
	
	`die letzte Teilmenge`
	
	`erste für den revolving`
	`door Algorithmus`
	`1 2 3 4 .... k`

Beispiel

source code	Eingabe	Ausgabe

`#include "_my.h"`
 	
`main()`
`{`
`int n,k;`
`_MY_SUBSET p;`
`scanf("%d",&n);`
`scanf("%d",&k);`
`_my_first_subset(n,k,&p);`
`do {`
`        _my_print_subset(&p);`
`} while (_my_next_k_subset(&p) == 0)`
`;`
`_my_free_subset(&p);`
`}`
	
	5
	2
		
		1 2
		2 3
		1 3
		3 4
		2 4
		1 4
		4 5
		3 5
		2 5
		1 5

Literatur

Martin Aigner: *Combinatorial Theory*, Springer 1979
F.J. Budden: *The Fascination of Groups*, Cambridge University Press 1972
Louis Comtet: *Advanced Combinatorics,* D. Reidel Publishing Company 1974
Gordon James, Adalbert Kerber: *The Representation Theory of the Symmetric Group,* Cambridge University Press (reprinted) 1989
Adalbert Kerber: *Algebraic Combinatorics via finite Group Actions*,BI Wissenschaftverlag 1991
Donald E. Knuth: *The Art of Computer Programming, Vol. 1/ Fundamental Algorithms*, 2nd Edition, Addison-Wesley 1973
Donald E. Knuth: *The Art of Computer Programming, Vol. 3/ Sorting snd Searching*, Addison-Wesley 1973
Albert Nijenhuis, Herbert S. Wilf: *Combinatorial Algorithms*, Academic Press 1978
Edward M. Reingold, Jurg Nievergelt, Narsingh Deo: *Combinatorial Algorithms: Theory and Practice*, Prentice Hall 1977
Richard P. Stanley: *Enumerative Combinatorics, Vol. 1*, Wadsworth 1986
Dennis Stanton, Dennis White: *Constructive Combinatorics*, Springer 1986
Herbert S. Wilf: *Combinatorial Algorithms: An Update*, SIAM 1989
S. Gill Williamson: *Combinatorics for Computer Science*, Computer Science Press 1985