A CLASS OF MULTINOMIAL PERMUTATIONS AVOIDING OBJECT CLUSTERS

RICHARD J. MATHAR

Abstract. The multinomial coefficients count the number of ways (of permutations) of placing a number of partially distinguishable objects on a line, taking ordering into account. A well-known two-parametric family of counts arises if there are objects of c distinguishable colors and m objects of each color, mc objects in total, to be placed on line.

In this work we propose an algorithm to count the permutations where no two objects of the same color appear side-by-side on the line. This eliminates all permutations with "clusters" of colors. Essentially we represent filling the line sequentially with objects as a tree of states where each node matches one partially filled line. Subtrees are merged if they have the same branching structure, and weights are assigned to nodes in the tree keeping track of how many mergers take place. This is implemented in a JAVA program; numerical results confirm Hardin's earlier counts for this kind of restricted permutations.

1. Unrestricted Multinomial Distributions

A fundamental counting argument considers N distinct objects and counts all distinct ways of placing them in a line. This is the number of permutations of N objects providing $N!$ possible arrangements. The rationale is that there are N choices to place an object at the leftmost position, this leaves $N-1$ choices to place an object of the remaining set at the next-to-left position, and so on, until only one possible choice is left to place the last object at the rightmost position.

If some of the objects are indistinguishable—and the distinction is made by color as usual in combinatorics—the pool of objects can be described by the notation $[m_1 m_2 \ldots m_c]$, meaning there are m_1 objects of the first color, m_2 objects of the second color and so on, with c different colors among the objects. The multiplicities m_i cause a reduction by the factors $m_i!$ relative to the count where all objects have different colors. The number of arrangements becomes the multinomial coefficient

$$\binom{N}{m_1 m_2 \ldots m_c} \equiv \frac{N!}{m_1! m_2! \cdots m_c!}, \quad N \equiv \sum_{i=1}^{c} m_i \tag{1}$$

2. Multinomials With Even Frequencies

If the pool of objects contains the same number of objects of some color with the same frequency m, the multinomial formula reduces to the simpler

$$\binom{N!}{mm \ldots m} \equiv \frac{(cm)!}{(m!)^c}, \quad N \equiv cm \tag{2}$$

Date: November 2, 2015.

2010 *Mathematics Subject Classification.* Primary 05A05; Secondary 68R05, 05C05.

Key words and phrases. Multinomial Distribution, Permutation, Nearest Neighbours.

$m\backslash c$	1	2	3	4	5
1	1	2	6	24	120
2	1	6	90	2520	113400
3	1	20	1680	369600	168168000
4	1	70	34650	63063000	305540235000
5	1	252	756756	11732745024	623360743125120

Table 1. Basic examples of the multinomial coefficients (2) [1, A089759,A060538].

$m\backslash c$	1	2	3	4	5
1	1	1	1	1	1
2	1	3	15	105	945
3	1	10	280	15400	1401400
4	1	35	5775	2627625	2546168625
5	1	126	126126	488864376	5194672859376

Table 2. Basic examples of the reduced multinomial coefficients $(mc)!/[(m!)^c c!]$ [1, A060540].

The associated counts with m varying from 1 to 5 down the rows and c varying from 1 to 5 along the columns are in Table 1. The first row at $m = 1$ are the factorials [1, A000142], restating the argument of the first paragraph in Section 1. If there is only a single color—represented by $c = 1$ and the first column—there is only a single arrangement. The entry at $c = m = 2$ for example counts chains of two colors (say `r` and `g`) occurring each twice; these are the 6 combinations `rrgg`, `rgrg`, `rggr`, `ggrr`, `grgr`, and `grrg`.

Combinations with a fixed set of colors are symmetric with respect to a permutation of the colors. The 6 combinations in the previous example with $c = 2$ colors can be constructed by taking the set of 3 combinations {`rrgg`, `rgrg`, `rggr`} and swapping `r` $\leftrightarrow$ `g` in each of these to generate the other 3 combinations.

Considering combinations equivalent which can be mapped onto each other by a permutation of the colors, the counts can be *reduced* by dividing them through $c!$. From Table 1 we arrive at Table 2. The 10 entries at $m = 3$ and $c = 2$ for example count the combinations `rrrggg`, `rrgrgg`, `rrggrg`, `rrgggr`, `rgrrgg`, `rgrgrg`, `rgrggr`, `rggrgg`, `rggrrg`, and `rggggr`, where the first place is forced to be an `r` to fix the reference color permutation.

3. Multinomials Without Clusters

3.1. **Definition of cluster avoidance.** Inspired by some sequences of Ron Hardin in the Online Encyclopedia of Integer Sequences [1], the main theme of this paper is to count multinomial combinations with equal frequencies as defined in Section 2 but enforcing that no two objects in the combinations that are neighbors have the same color —a sort of multi-sexual variant of the ménage seating problem with seats not cyclic around a table but arranged on a bench with two terminal chairs. The rule is that clusters of two or more objects with the same color in a run within the combinations must be avoided. Considering for example the $c = 3$ colors `r`, `g` and `b` each occurring $m = 2$ times, Table 2 counts 90 combinations like `rrggbb`, `rrgbgb`, `rgbrgb`, `rbgrgb` and so on; in the following we will for example not admit `rrggbb` with three clusters nor `gbrrbg` with one cluster. We denote the cluster-free combinations of mc objects with c colors each appearing m times by $M_{c,m}$

Definition 1. *(Cluster-free combinations with equal frequencies) $M_{c,m}$ is the number of combinations of cm objects—objects of c different colors appearing with frequency/multiplicities m—such that no two objects placed side-by-side in the combination have the same color.*

The restriction can only decrement the number of combinations relative to the unrestricted combinations:

$$M_{c,m} \leq \frac{(cm)!}{(m!)^c}. \tag{3}$$

The symmetry with respect to a permutation of the colors remains valid if non-clustering combinations are counted: A permutation of colors maps cluster-free chains of objects to cluster-free chains of objects and clustering chains of objects to clustering chains of objects. Therefore we may report results by tabulating the integers $M_{c,m}/c!$.

3.2. **Representation of Combinations as a Tree.** The following algorithm of obtaining concrete values of the $M_{c,m}$ was used in the program in the appendix. We lay down the objects left-to-right, and consider placing the next object as choosing a branch in the graphical representation of all combinations as a tree. At the root of the tree we still have all objects in a bag at our disposal, and there are c branches leading from there to the state after placing any of the objects with one of the c colors. Each of these states of the first generation splits into $c-1$ branches by placing one of the objects of the bag at the second position, where $c-1$ argues that any color but the one already placed may be chosen. The nodes generally split into $c-1$ branches, but the number of branches may be less if some of the colors are laid out early so there may be no more available from the bag further down in the tree. There are also leaves in the tree that are not the maximum distance cm away from the root, if only a set of objects of the same color remain in the bag.

The state of the $cm-g$ objects that are not yet placed in generation $0 \leq g$ may be put as a label at each node of the tree. One would use the vector $[m_1 m_2 \dots m_c]$ saying that there are m_1 objects of the first color, m_2 objects of the second color and so on yet to be placed, with $\sum_{i=1}^{c} m_i = cm - g$. In that way of accounting, the number of paths to the leafs in generation $g = cm$ is $M_{c,m}$.

3.3. **Subtree Mergers.** Next we reduce the tree to a (usually higher connected) graph by combining all nodes at each individual generation with the same subtree structure into a single node, and give that node a weight w equivalent to the number of all paths leading to the individual nodes of the sparse original tree.

Define a label at a node in the merged tree as $(0^{f_0}1^{f_1}\cdots m^{f_m}r_p^{1'} : w)$. It represents that there are $0\times f_0 + 1\times f_1 + \cdots m\times f_m + r_p \times 1$ objects in the bag not yet placed in the path from the root of the tree up to and including that node. f_0 of the colors are not in the bag, f_1 of the colors have one object remaining in the bag, f_2 of the colors have two objects remaining in the bag and so on. The frequencies f do *not* include the color that cannot be placed next to avoid clustering; *that* color is represented by the last entry tagged with a prime representing one color with r_p remaining objects in the bag not to be placed next, where $0 \leq r_p \leq m$.

This structure information suffices to compute all possible labels of the next generation and also the full branching choices.

Remark 1. *For this purpose we will actually omit all elements j^{f_j} in the notation where either $j = 0$ or $f_j = 0$, because branches cannot be generated by taking objects that are no longer in the bag.*

The transition from one generation to the next means to place another object. This implies to select (in a loop) one of the non-primed subsymbols j^{f_j} in the label, $j > 0$ and $m_j > 0$, picking one of the object colors and object of which at least one remains in the bag. In the next generation the number colors in the bag that have j objects will be one less, so j^{f_j} is replaced by j^{f_j-1}. The color that was picked replaces the previous primed color, but with one object less remaining that has been placed, so $r_p^{1'}$ is replaced by $(j-1)^{1'}$. The previously primed color returns (unprimed) into the bag: $r_p^{1'}$ becomes r_p^1. If the new list contains the same j more than once, they are merged by adding their f_m. The old weight w is replaced by the product wf_j because there are f_j choices of selecting a color of type j^{f_j} in the bag.

In a double loop over all symbols j^{f_j} in each label and over all labels of some generation, a new set of symbols is created representing the next generation. This is condensed by another merging of labels in the new generation which have the same set of subsymbols in front of the colon: the merged label has the same subsymbols and a weight which is the sum of the weights of the individual labels.

3.4. **Example.** We illustrate that efficient way of accounting in all generations starting with a bag of $c = 3$ colors, each color with $m = 3$ objects. The number of equivalent nodes in the sparse (non-merged) version of the tree is the sum of all weights of all labels in a generation, and listed as N. In generation 0 (before having placed any object), we have three colors each with 3 objects, and no primed color (because there is no restriction to avoid clusters yet). The label is

```
(3^3 :1)
gen 0 N= 1
```

In generation 1, one of the colors is selected, so the multiplicity of the number of colors with 3 objects drops from 3 to 2, and the color of the selected object (2 objects remaining) moves into the primed section. The weight is the old weight multiplied by the number of choices of the colors, 1×3. The label in generation 1 becomes

```
(3^2 2^1' :3)
gen 1 N= 3
```

In generation 2 we select one of the 2 colors each with 3 objects remaining, so 3^2 becomes 3^1 and the selected color becomes $2^{1'}$ in the primed subsymbol. The primed $2^{1'}$ returns as 2^1 into the unprimed list. The weight is the old weight multiplied by the multiplicity of (equivalent) colors to be placed 3×2:

```
(2^1 3^1 2^1' :6)
gen 2 N= 6
```

In generation 3 we have two choices of picking a color class: one from the subsymbol 2^1 in the label and another from the subsymbol 3^1. The first leaves 3^1 untouched, moves 2^1 as $1^{1'}$ into the primed part, and moves $2^{1'}$ to 2^1, with a weight 1×6. The second leaves 2^1 untouched, moves 3^1 as $2^{1'}$ into the primed part, and moves $2^{1'}$ to 2^1, with a weight 1×6. This creates two kinds of bags in generation 3, $(2^1 3^1 1^{1'})$ and $(2^2 2^{1'})$:

```
(2^1 3^1 1^1' :6)(2^2  2^1' :6)
gen 3 N= 12
```

In generation 4 we can pick 2^1 from the first bag and generate $(1^1 3^1 1^{1'} : 6)$, we can pick 3^1 from the first bag and generate $(1^1 2^1 2^{1'} : 6)$, or we can pick 2^2 from the second bag and generate $(2^2 1^{1'} : 12)$. So far we have generated $N = 6+6+12 = 24$ different partial chains:

```
(1^1  3^1 1^1' :6)(1^1 2^1  2^1' :6)(2^2  1^1' :12)
gen 4 N= 24
```

In generation 5

(1) we can pick 1^1 from the first bag and generate $(1^1 3^1 0^{1'} : 6)$;
(2) we can pick 3^1 from the first bag and generate $(1^2 2^{1'} : 6)$. Here the 3^1 becomes 3^0 and $2^{1'}$. 3^0 is removed, and the 1^1 which remains and the 1^1 which is unprimed are merged into 1^2;
(3) we can pick 1^1 from the second bag and generate $(2^2 0^{1'} : 6)$;
(4) we can pick 2^1 from the second bag and generate $(1^1 2^1 1^{1'} : 6)$. Here 2^1 becomes 2^0 and $1^{1'}$. 2^0 is removed. The 1^1 remains. The $2^{1'}$ is unprimed and becomes 2^1;
(5) we can pick 2^2 from the third bag and generate $(1^1 2^1 1^{1'} : 24)$. Here 2^2 becomes 2^1 and $1^{1'}$. The $1^{1'}$ is unprimed and becomes 1^1. The old weight 12 is multiplied by the "exponent" 2 and becomes 24;

In picks (4) and (5) the new bags $(1^1 2^1 1^{1'} : 6)$ and $(1^1 2^1 1^{1'} : 24)$ have the same labels apart from their weights and can be merged into $(1^1 2^1 1^{1'} : 30)$ by adding weights. This leads to 4 types of bags representing $6+6+6+30 = 48$ nodes in the sparse tree:

```
(1^1  3^1 0^1' :6)(1^2   2^1' :6)( 2^2  0^1' :6)(1^1 2^1  1^1' :30)
gen 5 N= 48
```

In generation 6 we generate 5 different types of bags representing 96 nodes in the sparse tree:

```
(   3^1 0^1' :6)( 1^1   2^1' :6)(1^1 2^1  0^1' :42)(  2^1  1^1' :12)(1^2   1^1' :30)
gen 6 N= 96
```

In generation 7 we generate 4 different types of bags representing 168 nodes in the sparse tree:

```
(    2^1' :6)(  2^1  0^1' :48)( 1^1   1^1' :54)(1^2   0^1' :60)
gen 7 N= 168
```

In generation 8 we generate 2 different types of bags representing $48+120 = 222$ nodes in the sparse tree. There is no output/branch from $(2^{1'} : 6)$. $(2^1 0^{1'} : 48)$ generates $(1^{1'} : 48)$. $(1^1 1^{1'} : 54)$ generates $(1^1 0^{1'} : 54)$. $(1^2 0^{1'} : 60)$ generates $(1^1 0^{1'} : 120)$. The last two of these new bags are merged:

```
(    1^1' :48)( 1^1   0^1' :174)
gen 8 N= 222
```

In generation 9 we observe that no branches emerge from the label $(1^{2'} : 48)$ because there are not objects left in the unprimed list. That label in generation 8 does not generate anything in generation 9. The label $(1^1 0^{1'} : 174)$ has one choice of a single object and generates

```
(     0^1' :174)
gen 9 N= 174
```

In the listing of results this is divided by $c! = 3! = 6$ to account for the permutation of the colors, so $M_{3,3}/6 = 29$.

In some sort of pure book-keeping we observe that in generation 10 (which is larger than cm) there are no chains of objects of that type (because after generation 9 no objects are left in the bag):

```
gen 10 N= 0
```

3.5. **Overview of the Program.** A label in the algorithm is represented by an object (in the OO-sense) of the class `RemState` (State of the colored objects remaining in the bag). This contains a weight w and a list of j^{f_j} that are instances of the `MultState` class. A `MultState` object holds the number j and its multiplicity f_j and a boolean flag which indicates whether this is a primed symbol $j^{f_j}{}'$ or not.

An object of the `RemState` class can be printed in an ASCII format with the `toString` function, which is useful to track the transformations as shown in the previous example. It can be transformed into a state of the next generation by placing one of its objects, which is done by calling the `place` function with an 0-based index into the symbols j^{f_j} to detail which type of colors is to be picked for the next placement. All possible new states (branches in the tree) derived from the label are created with the `branches` function which basically loops through all the symbols j^{f_j} and merges the new states wherever possible.

The class `StateVector` is a collection of the labels, equivalent to a collection of all nodes (labels) in the tree at the same distance from the root, and equivalent to all states of the same generation. A call to `nextgen` creates all states of the next generation by calling the `branches` function of its individual labels and merging the new labels where possible.

The `main` function of the class initializes the `StateVector` with a single node equivalent to the root of the tree by using the c and m taken from the command line. It calls recursively the `nextgen` function cm times to place all objects, and eventually reports the total weight $M_{c,m}/c!$ at the leaf of the merged tree.

4. Numerical Results

If we comple and run the program of the appendix with a double loop over m and c like

```
#!/usr/bin/env bash

javac *.java >& /dev/null
for m in {1..10} ; do
        for c in {1..10} ; do
                java -cp . StateVector -q $c $m ;
        done
        echo
done
```

we obtain the following table of results. Each line shows c, then m and then the reduced $M_{c,m}/c!$ as if reading the array down the columns:

```
1 1 1
2 1 1
3 1 1
4 1 1
5 1 1
6 1 1
7 1 1
8 1 1
9 1 1
10 1 1

1 2 0
2 2 1
3 2 5
4 2 36
5 2 329
6 2 3655
7 2 47844
8 2 721315
9 2 12310199
10 2 234615096

1 3 0
2 3 1
3 3 29
4 3 1721
5 3 163386
6 3 22831355
7 3 4420321081
8 3 1133879136649
9 3 372419001449076
10 3 152466248712342181

1 4 0
2 4 1
3 4 182
4 4 94376
5 4 98371884
6 4 182502973885
7 4 551248360550999
8 4 2536823683737613858
9 4 16904301142107043464659
10 4 156690501089429126239232946

1 5 0
2 5 1
3 5 1198
4 5 5609649
5 5 66218360625
6 5 1681287695542855
7 5 81644850343968535401
8 5 6945222145021508480249929
9 5 967335448974819561548523580438
```

10 5 209141786137614009701487336108267723

1 6 0
2 6 1
3 6 8142
4 6 351574834
5 6 47940557125969
6 6 16985819072511102549
7 6 13519747358522016160671387
8 6 21671513613423101256198918372909
9 6 64311863997340571475504065539218471107
10 6 330586922756304429697714946501284146322953006

1 7 0
2 7 1
3 7 56620
4 7 22875971289
5 7 36533294879349056
6 7 183095824753841610373405
7 7 2421032324142610480402567434373
8 7 74115215422015289392187745053216373265
9 7 4749303210651587675797285013227098386984170468
10 7 588242979144354234332728292738493758656488275002948671

1 8 0
2 8 1
3 8 400598
4 8 1530622143864
5 8 28920026907938624194
6 8 2070756746775910218326948065
7 8 459408385876250801291447710561829082
8 8 271259741131895052775392614041761701799270286
9 8 379065045836307787068046364731543393514652159389593652
10 8 1142490930667808363833513276790270503563005942443336645995266946

1 9 0
2 9 1
3 9 2872754
4 9 104650147201049
5 9 23575497690601916022516
6 9 24302858067615766089801166488125
7 9 91155245844064069307740171414201519055298
8 9 1046031892354833895113128900608177633584652958677057
9 9 32119646666355552112999645991677870426882424139287301894021793
10 9 2373613014676717426115059968329689897428343332868848727660084375739772331

1 10 0
2 10 1
3 10 20824778
4 10 7279277647839552
5 10 19672658572012343899666292
6 10 293736218147318801678882792470437721
7 10 18739368045280595665934917472507368174737872589

```
8 10 4204427313459831775866154680419213479057724331798640498651
9 10 2853894650868039917308960659804624462496672564449632072749348382989169
10 10 5202145573917369427233280073392338673677947120503038749310295939572928744511760680
```

The leading group where $m = 1$ simply states that the number of permutations of distinct objects equals the factorial of the number of objects: $M_{c,1} = c!$. If this number is reduced by dividing through the factorial,

$$M_{c,1}/c! = 1. \tag{4}$$

The first line in each group states that $M_{1,m}$ is mostly zero, the multiplicity $m = 1$ being the only exception:

$$M_{1,m} = \delta_{1,m} \tag{5}$$

This is trivial and means that if there is only one color, placing the objects in a line is impossible if no neighbors of the same color are allowed and if there is more than one object.

The second line in each group reports another obvious result. If the number of colors is $c = 2$, the objects must be arranged in a list of alternating colors, and there are only the two choices of beginning with either color:

$$M_{2,m}/2! = 1. \tag{6}$$

The second group at $m = 2$ appears to be absolute values of Bessel polynomials evaluated at -1 [1, A000806][2, (15)] with recurrence

$$\frac{M_{c,2}}{c!} + (1-2c)\frac{M_{c-1,2}}{(c-1)!} - \frac{M_{c-2,2}}{(c-2)!} = 0. \tag{7}$$

for the reduced counts, respectively

$$M_{c,2} - (2c-1)(c-1)M_{c-1,2} - (c-1)(c-2)M_{c-2,2} = 0. \tag{8}$$

The groups with $m = 3, \ldots 7$ confirm Hardin's results [1, A190826, A190830, A190833, A190835, A190836].

One can read the array $M_{c,m}/c!$ in the transposed format along the lines and obtains the sequences $1, 5, 29, 182, 1198, \ldots$ [1, A190917], $1, 36, 1721, 94376, \ldots$ [1, A190918], $1, 329, 163386, 98371884, \ldots$ [1, A190920], [1, A190923,A190927,A190932].

Appendix A. JAVA Source Code

A.1. **MultState.java.**

```java
/* package de.mpg.mpia.rjm
*/

import java.util.* ;

/**
* An object of the class represents a symbol leftN^leftFreq with an optional
* prime (if tagged is true).
* @author R. J. Mathar
* @since 2015-11-02
*/
public class MultState implements Comparable<MultState>
{
    /** True if tagged with a prime (not to be placed in next placement)
```

```
    */
    public boolean tagged ;

    /** Remaining items for further placement. The "base" number of the symbol.
    */
    public int leftN ;

    /** Number of colors that have leftN items left to distribute.
    * The "exponent" of the symbol.
    */
    public int leftFreq ;

    /** Ctor.
    * Generate a symbol with all three parameters known.
    * @param numbleft Number of items let in that color class.
    * @param multipl Number of color classes with that (same) numbleft.
    * @param forbid True if this color has just been placed.
    *    True implies that multipl=1.
    */
    public MultState(int numbleft, int multipl, boolean forbid)
    {
        leftN=numbleft ;
        leftFreq = multipl ;
        tagged = forbid ;
    } /* ctor */

    /** Represent the symbol in ASCII art for logging purposes.
    * @return The base number leftN followed by the caret, the exponent  (leftFreq) and the prime (if t
    */
    public String toString()
    {
        String str  = new String() ;
        /* there is a long notation which prints all states, including those with leftN or leftFreq=0,
        * and a shorter, which skips these; we save some paper and preselect to get the shorter.
        */
        if ( leftN > 0 && leftFreq >0 || tagged )
        {
            str += leftN + "^" + leftFreq ;
            if (tagged)
                str += "'" ;
        }
        return str ;
    }

    /** A ranking function of the subsymbols.
    * In a label in the tree graph, the state is basically independent on
    * the ordering of the symbols. So we may keep them sorted for faster insertion
    * and merger operations. In a somewhat arbitrary fashion we consider the primed
    * part the largest (rightmost in the paper), and sort the other ones first with
    * respect to the base (leftN) and second with respect to the exponent (leftFreq).
    * The main outcome is that if all subsymbols (not considering the weight) in a label
    * are the same after ordering, the labels can be merged by adding their weights.
    * @return -1, 0 or 1 depending on whether this is considered smaller than, equal to or larger than
```

```
    */
    public int compareTo(MultState oth)
    {
        /* consider tagged states to be larger
        */
        if ( tagged && ! oth.tagged)
            return 1;
        else if ( ! tagged && oth.tagged)
            return -1;
        else
        {
            /* same tagged class. consider larger if items left larger
            */
            if ( leftN > oth.leftN)
                return 1;
            else if ( leftN < oth.leftN)
                return -1;
            else
            {
                if ( leftFreq > oth.leftFreq)
                    return 1 ;
                else if ( leftFreq < oth.leftFreq)
                    return -1 ;
                else
                    return 0 ;
            }
        }
    } /* compareTo */
} /* MultState */
```

A.2. **RemState.java.**

```
/* package de.mpg.mpia.rjm
*/

import java.util.* ;
import java.math.* ;

/** A RemState object is a label, a bag of remaining objects with a weight.
* @author R. J. Mathar
* @since 2015-11-02
*/
public class RemState implements Comparable<RemState>
{
    /** The remaining choices of states, sorted in increasing order left (smallest index) to right.
    * This will usually contain exactly one state with a primed subsymbol, unless this
    * is the root of the tree where no color is in the forbidden list.
    */
    public Vector<MultState> state ;

    /** Number of occurrences and the combinatorial number of paths to that state.
    * The piece in the label after the colon, representing the node count in the sparse representation
    * of the tree of placements.
    */
```

```
    public BigInteger weight ;

    /** ctor representing an impossible state.
    * The weight and the total number of items in the bag are set to zero.
    */
    public RemState()
    {
        state = new Vector<MultState>() ;
        weight = BigInteger.ZERO ;
    } /*ctor */

    /** ctor with initial state of c colors each multiplicity m.
    * The total number of items is c times m and there is no tagged (forbidden) color yet.
    * This is the bag at generation zero at the root of the tree, where all objects
    * are still to be distributed and none has yet been placed.
    * @param c the number of different colors of the items
    * @param m the multiplicity. The number of items with the same color.
    */
    public RemState(int c, int m)
    {
        state = new Vector<MultState>() ;
        state.add(new MultState(m,c,false)) ;
        weight = BigInteger.ONE ;
    } /*ctor */

    /** ctor with initial state of c colors each multiplicity m
    * The total number of items is c times m.
    * @param c the number of different colors of the items
    * @param m the multiplicity. The number of items with the same color.
    * @param w the combinatorial weight.
    * @obsolete not needed
    public RemState(int c, int m, BigInteger w)
    {
        state = new Vector<MultState>() ;
        state.add(new MultState(m,c,false)) ;
        weight = w ;
    }
    */

    /** A ASCII representation of the contents of this type of bag.
    * @return The list of subsymbols followed by a colon and the weight surrounded by parenthesis.
    */
    public String toString()
    {
        String str = new String() ;
        str += "(" ;
        for ( MultState s: state)
            str += s.toString() + " " ;
        str += ":" + weight + ")" ;
        return str ;
    }

    /** Select an object by index in the state vector and place it.
```

```
    * @param itmNo A number from 0 to state.size()-1.
    * @return The state in the next generation after itmNo has been placed.
    *   If this cannot be done because itmNo points to a subsymbol that does not
    *   exist or points to a subsymbol with no objects left, this state may be the empty state.
    */
    public RemState place(int itmNo)
    {
        /* Index out of bounds, meaning we are pointing not to a subsymbol.
        */
        if (itmNo <0 || itmNo >= state.size() )
            return new RemState() ;

        /* for shorter notation below: a reference to the selected subsymbol.
        */
        final MultState modf = state.elementAt(itmNo) ;

        /* Cannot place the tagged item, either because this is in the forbidden list
        * or because there are no objects left in that subsymbol's class.
        */
        if ( modf.tagged || modf.leftFreq <= 0 || modf.leftN <= 0)
            return new RemState() ;

        /* Place the element (and copy the weight).
        * Virtually split off remaining elements (which get frequency one less)
        * (itms0^colno0, itm1^colno1,  itm2^colon2, ..itmi^colnoi, itmtagge^1:...)
        * becomes
        * (itms0^colno0, itm1^colno1,  itm2^colon2, ..itmi^(colnoi-1), itmi^1, ..., itmtagge^1:...).
        * The previously primed/tagged item returns to the non-primed elements (and optionally
        * merged with an existing one if possible), and the new primed element is
        * constructed as nextTagged.
        */
        MultState nextTagged = new MultState(modf.leftN-1,1,true) ;

        /* nextUnTagged is what results if the currently tagged/primed subsymbol is
        * merged into the pool of untagged states. We search for the primed/tagged
        * subsymbol (although just picking the last should suffice if that is not
        * the root tree), and toggle its tagged flag.
        */
        MultState nextUnTagged = null ;
        for( MultState s : state)
        {
            if ( s.tagged)
            {
                /* toggle the tag; keep the number of objects with that particular color.
                */
                nextUnTagged = new MultState(s.leftN,1,false) ;
                /* there should be only one tagged state, so we leave the loop
                * early if that one is detected.
                */
                break;
            }
        }

```

```
/* Compose the new state vector in increasing order by moving upwards
* through the existing elements, merging in or copying in the nextUnTagged
* where the ordering defines its place in the vector of subsymbols.
*/
RemState deriv = new RemState() ;

/* The weight of the new bag is the old weight times the multiplicity of the colors
* associated with index itmNo.
*/
deriv.weight = weight.multiply(new BigInteger(""+modf.leftFreq)) ;

/* The new state vector composed by scanning the existing subsymbols
* of the previous bag in order.
*/
for( int i=0 ; i < state.size() ; i++)
{
    MultState s = state.elementAt(i) ;
    /* if this was the type of objects out of which itmNo was taken: the actual residual
    * frequency has diminished by 1. This has not changed the position of
    * this s in the vector because we are using the "base" of the subsymbol
    * as the major ordering.
    */
    if ( i == itmNo)
        /* Here s.tagged is false, because those cases returned already above
        * where the modf.tagged cases returned the empty state.
        */
        s = new MultState(s.leftN, s.leftFreq-1,false) ;

    /* reinsert any non-tagged s; the new tagged/primed one will
    * be appended after this loop over the i is finshed. Instead of
    * admitting the subsymbol with s.tagged =true we insert the
    * nextUnTagged constructed above.
    */
    if ( ! s.tagged)
    {
        if ( nextUnTagged != null)
        {
            /* take care of inserting and/or merging the previously tagged subsymbol
            */
            if ( s.leftN < nextUnTagged.leftN )
            {
                /* not yet reached the position of inserting nextUnTagged.
                */
                deriv.state.add(s) ;
            }
            else if ( s.leftN == nextUnTagged.leftN )
            {
                /* The place of merger of prviously tagged and untagged symbols.
                * The frequence is the sum of the frequencies and they have a common
                * "base" in the subsymbol notation.
                */
                MultState merg = new MultState(s.leftN,s.leftFreq+nextUnTagged.leftFreq,false)
                /* do not put s but the merged subsymbol into the new label
```

```
                        */
                        deriv.state.add(merg) ;
                        /* Invalidate the nextUnTagged to indicate to the trigger
                        * right after the loop that this is already dealt with.
                        */
                        nextUnTagged = null ;
                    }
                    else
                    {
                        /* given state larger: two cases for the old primed label, either already merge
                        * the untagged state or the untagged state must be inserted prior to s.
                        */
                        if ( nextUnTagged != null)
                        {
                            deriv.state.add(nextUnTagged) ;
                            nextUnTagged = null ;
                        }
                        deriv.state.add(s) ;
                    }
                }
                else
                    deriv.state.add(s) ;
            }
        }

        /* If not yet merged inside the previous loop drop the nextUnTagged subsymbol
        * at the end of the label.
        */
        if ( nextUnTagged != null)
            deriv.state.add(nextUnTagged) ;

        /* by the choice of ordering, the new primed/tagged subsymbol goes last in the list
        */
        deriv.state.add(nextTagged) ;

        return deriv ;
    } /* place */

    /** Distance to the leaf states of the tree.
    * @return The sum over the products of leftN and leftFreq over all terms in state.
    * @obsolete Not needed
    public int distToLeaf()
    {
        int dist =0 ;
        for( MultState s : state)
            dist += s.leftN * s.leftFreq ;
        return dist ;
    }
    */

    /** The descendants in the next generation derived by placing any one item.
    * @return A new set of bags that are created by considering all unprimed subsymbols.
    *  This may be an empty vector if there are no ways to place another object taken from the current
```

```
    */
    public Vector<RemState> branches()
    {
        Vector<RemState> br = new Vector<RemState>() ;
        /* The loop of all non-tagged items considered for new types of bags.
        */
        for(int itmIdx =0 ; itmIdx < state.size () ; itmIdx++)
        {
            /* this is the type of bag created with that selection in the new generation
            */
            RemState child = place(itmIdx) ;

            /* zero weight means there was no way to place another object.
            * dismiss these cases.
            */
            if ( child.weight.compareTo(BigInteger.ZERO) > 0 )
            {
                /* Search in the already generated types of bag if this is
                * already produced. If yes then merge, adding weights
                */
                for( RemState given : br)
                {
                    if (  given.compareTo(child) == 0 )
                    {
                        /* same list of subsymbols: can merge them
                        * and delete child to indicate to the post-loop decision
                        * that this is already dealt with.
                        */
                        given.weight = given.weight.add(child.weight) ;
                        child = null ;
                        break;
                    }
                }
                /* if not mergeable with a label already in the list:
                * add it to the labels list to be returned.
                */
                if ( child != null)
                    br.add(child) ;
            }
        }
        return br ;
    } /* branches */

    /** Comparator with respect to similarity of the subsymbols.
    * @param oth The label this is to be compared with.
    * @return 1 if in a left-right comparison of the states this here is
    *    larger than oth, 0 if equal, -1 if less.
    */
    public int compareTo(RemState oth)
    {
        /* on a rough scale order by the number of subsymbols
        */
        if ( state.size() > oth.state.size() )
```

```
            return 1;
        else if ( state.size() < oth.state.size() )
            return -1;
        else
        {
            /* Assume that inside the list state of subsymbols the
            * items are already ordered,
            * so we do not need to sort the state first.
            */
            for (int s =0 ; s < state.size() ; s++)
            {
                if ( state.elementAt(s).compareTo( oth.state.elementAt(s)) > 0 )
                    return 1;
                else if ( state.elementAt(s).compareTo( oth.state.elementAt(s)) < 0 )
                    return -1;
            }
            /* if we arrive here the number and type of all subsymbols are equal.
            * This indicates to RemState.branches() that we can merge this with oth.
            */
            return 0 ;
        }
    } /* compareTo */
} /* RemState */
```

A.3. **StateVector.java.**

```
import java.util.* ;
import java.math.* ;

/** An object of the StateVector type is a collection of labels representing all merged nodes of a gene
* @author R. J. Mathar
* @since 2015-11-02
*/
public class StateVector
{
    /* the collection of all labels in this generation
    */
    public Vector<RemState> generat ;

    /** ctor that creates the unique state of the 0'th generation, label (c^m :1)
    * @param c Number of different colors of the objects.
    * @param m Multiplicity. The number of objects with the same color.
    */
    public StateVector(int c, int m)
    {
        /* put exactly one state with label (c^m:1) into the set of labels/bags
        */
        generat = new Vector<RemState>() ;
        RemState root = new RemState(c,m) ;
        generat.add(root) ;
    } /* ctor */

    /** ctor given a known set of labels/bags.
    * @param stats The collection of labels to be considered.
```

```
    * Note that there are no consistency checks that the stats are all of the
    * the same generation. [Basically one would check that the sum of all subsymbols
    * (sum over products of base and exponent, primed or not) is the same number for
    * all elements in stats.]
    */
    public StateVector(Vector<RemState> stats)
    {
        generat = stats ;
    } /* ctor */

    /** A representation as a set of parenthesis, each parentheses a label.
    * @return The set of bags/labels in the form (...)(....).
    */
    public String toString()
    {
        String str = new String() ;
        for ( RemState s : generat)
            str += s.toString() ;
        return str ;
    } /* toString */

    /** Factorial of a positive integer
    * @param n The argument of the factorial .
    * @return n! The product 1*2*3*4*...*n.
    */
    static public BigInteger factorial(int n)
    {
        BigInteger f = BigInteger.ONE ;
        for (int i=2 ; i <= n ; i++)
            f = f.multiply(new BigInteger(""+i) ) ;
        return f;
    } /* factorial */

    /* Graph-theoretical distance to the leafes of the tree.
    * equivalent to the number of objects yet to be placed.
    * @obsolete Not needed
    public int distToLeaf()
    {
        * this number is the same for all elements in the generation,
        * so we can get it from the first element (which exists)
        return generat.firstElement().distToLeaf() ;
    }
    */

    /** Combinatorial weight: sum over all weights in all bags/labels.
    * @return The number of combinations admitted up to this generation.
    */
    public BigInteger Ncombinat()
    {
        BigInteger w = BigInteger.ZERO ;
        for( RemState s : generat)
        {
            w = w.add(s.weight) ;
```

```
        }
        return w;
    } /* Ncombinat */

    /** Generate the next generation equivalent to placing one more object at all current leafs of the
    * @return The set of bags/labels that are created if all branches of all
    *   bags/lables in this state are created.
    * This vector of new labels is condensed by merging labels with the same subsymbol set.
    */
    public StateVector nextgen()
    {
        /* children states, the next generation where each back has one object less than this
        */
        Vector<RemState> childr = new Vector<RemState>();
        /* for each of the labels/bags : construct all branches (new states)
        */
        for( RemState s : generat)
        {
            /* generate all new labels from that particular label/bag
            */
            Vector<RemState> sderiv = s.branches() ;

            /* try to merge them into the existing set of labels, superimposing branches.
            * To this purpose scan the already known types of bags in childr
            * and compare their signatures for each label just created.
            */
            for( RemState nxt : sderiv)
            {
                /* flag that nxt was not yet found in childr and not yet merged
                */
                boolean mrgd = false ;
                for( RemState given : childr)
                {
                    if ( given.compareTo(nxt) == 0 )
                    {
                        given.weight = given.weight.add(nxt.weight) ;
                        mrgd = true;
                        break ;
                    }
                }
                /* append the label derived from s to the labels of the next generation
                */
                if ( ! mrgd)
                    childr.add(nxt) ;
            }
        }

        return new StateVector(childr) ;
    } /* nextgen */

    /** Main executable function.
    * Usage:
    *  javac *.java
```

```
 *  java -cp . StateVector [-q] c m
 * where the optional switch -q leads to a quieter output, which means that the
 * explicit printing of all the labels in all generations is skipped.
 * The two positive integer parameters c and m are the number of colors
 * and the number of objects with the same color (multiplicity/frequence of each
 * color). The number of generations to be constructed is the product c*m.
 * The output (if -q is given) is c followed by a blank and m followed by
 * a blank and the M_{c,m}/c! number of permutations inequivalent under the
 * permutation of the colors.
 * @author R. J. Mathar
 * @since 2015-11-02
 */
public static void main(String[] args)
{
    if ( args.length < 2 )
    {
        System.err.println("Usage: StateVector [-q] #colors #multiplic") ;
        System.exit(1) ;
    }
    int optind=0 ;

    /* verbosity on by default, but switched off if -q in the argument list.
    */
    boolean verb=true ;
    if ( args[optind].compareTo("-q") == 0 )
    {
        verb = false;
        optind++ ;
    }

    /* collect the two parameters c and m from the command line.
    */
    int c = (new Integer(args[optind++])).intValue() ;
    int m = (new Integer(args[optind++])).intValue() ;

    /* degenarcy factor c! with respect to permutations of the colors
    */
    BigInteger cDegen = factorial(c) ;

    /* construct the top of the tree, generation 0
    */
    StateVector curr = new StateVector(c,m) ;

    /* loop over all generations, of which there are c*m more
    */
    for(int g=0 ; g <= 1+c*m; g++)
    {
        /* count the sum of the weights (number of combinations) in that generation
        * and divide by c!
        */
        BigInteger canDegen = BigInteger.ZERO ;
        if ( curr.Ncombinat().mod(cDegen).compareTo(BigInteger.ZERO) == 0 )
            canDegen = curr.Ncombinat().divide(cDegen) ;
```

```

            if ( verb)
            {
                /* if verbose, print all labels in the current generation
                */
                System.out.println(curr);
                System.out.print("gen " + g + " N= " + curr.Ncombinat() );
                if ( g>= 2 && canDegen.compareTo(BigInteger.ZERO) > 0 )
                    System.out.println(" reduced by " + cDegen + " " + canDegen );
                else
                    System.out.println();

                System.out.println() ;
            }
            else if ( g == c*m)
            {
                /* reached the leaf of the tree, so report the number of combinations
                */
                System.out.println(c + " " + m + " " + canDegen);
            }

            /* consider this a stack of 2 generations. Generate the next generation in nxt,
            * and replace the current generation by this.
            */
            StateVector nxt = curr.nextgen() ;
            curr = nxt ;
        }
    } /* main */
} /* StateVector */
```

References

1. Neil J. A. Sloane, *The On-Line Encyclopedia Of Integer Sequences*, Notices Am. Math. Soc. **50** (2003), no. 8, 912–915, http://oeis.org/. MR 1992789 (2004f:11151)
2. Jacques Touchard, *Nombres exponentiels et nomber de bernoulli*, Can. J. Math. **8** (1956), 305–320.

URL: `http://www.mpia.de/~mathar`

Max-Planck Institute of Astronomy, Königstuhl 17, 69117 Heidelberg, Germany