CORRIGENDUM TO "POLYOMINO ENUMERATION RESULTS. (PARKIN ET AL., SIAM FALL MEETING 1967)"

RICHARD J. MATHAR

Abstract. This work provides a Java program which constructs free polyominoes of size n sorted by width and height of the convex hull (i.e., its rectangular bounding box). The results correct counts for 15-ominoes published in the 1967 proceedings of the SIAM Fall Meeting, and extend them to 17-ominoes and partially to even larger polyominoes. [vixra:1905.0474]

1. Free polyominoes

1.1. **Nomenclature.** Polyominoes are sets of n edge-connected squares, which means each of the squares can be reached from any other square of the set by a path that connects nearest neighbors (adjacent squares to the North, East, South and West) which all are members of the set.

Fixed polyominoes are polyominoes which can be mapped onto each other by translating them rigidly along the horizontal and/or vertical axes of the underlying square grid. They can be enumerated for example with the aid of a transfer matrix method [3][2, A001168][1].

A set of fixed polyominoes that can be mapped onto each other by further symmetry operations of rotations by multiples of 90 degrees and/or flips along the horizontal or vertical axes is a *free* polyomino. The convex hull (or bounding box) of a polyomino is the largest connected rectangular section of the underlying square grid such that each row and column of the hull contains at least one square of the polyomino. We are interested in classifying all n-ominoes by the height h and width w of their convex hull.

Definition 1. *(Free Polyominoes Classified by Bounding Box) $P_{h\times w}(n)$ denotes the number of free polyominoes of size n that fit into a tight $h \times w$ bounding box.*

The number of free polyominoes does not change if the bounding rectangle is rotated by multiples of 90 degrees (such that the roles of width and height are swapped):

$$P_{h\times w}(n) = P_{w\times h}(n). \tag{1}$$

There is only one free block polyomino where all cells within the bounding box of area hw are occupied:

$$P_{h\times w}(hw) = 1. \tag{2}$$

Date: March 25, 2021.
2010 *Mathematics Subject Classification.* Primary 05B50; Secondary 05A18, 51-04.
Key words and phrases. Free Polyomino, Enumeration, Convex Hull.

There is no polyomino if the number of cells in the bounding box is smaller than the number of cells in the polyomino:

$$P_{h\times w}(n) = 0, \text{ if } n > hw. \tag{3}$$

There is no polyomino if the width and height of the bounding box are too large to connect all cells of the n-omino:

$$P_{h\times w}(n) = 0, \text{ if } h + w - 1 > n. \tag{4}$$

Definition 2. *(Free Polyominoes) The total number of free n-ominoes is*

$$P(n) \equiv \sum_{h=1}^{n}\sum_{w=1}^{h} P_{h\times w}(n). \tag{5}$$

The perimeter p of a polyomino is the number of polyomino edges which are not shared by two polyominoes. (These include *inner* edges of polyominoes with holes.) Because the native number of edges in a monomino is 4, the number could be computed as $p = 4n-2E$, where E is the number of edges in the polyedge. (The polyedge is the graph where each occupied cell is a node and nodes are connected if the cells are neighbors along one of the 4 directions. The degree of nodes in the polyedge is at most 4.)

Definition 3. *(Free Polyominoes classified by Bounding Box and Perimeter) $P_{h\times w,p}(n)$ denotes the number of free polyominoes of size n and perimeter p that fit into a tight $h \times w$ bounding box.*

If we look from each of the 4 major directions at the periphery of a polyomino, we see at least one edge of the outer cells, but not the edges in cavities and holes, so a lower bound is

$$P_{h\times w,p}(n) = 0 \quad \text{if} \quad p < 2w + 2h. \tag{6}$$

Another bound is [9, 8]

$$P_{h\times w,p}(n) = 0 \quad \text{if} \quad p < 2\lceil 2\sqrt{n}\rceil. \tag{7}$$

The number E of edges in the polyedge is limited by the requirement of connectivity of the cells, so the "thin" tree-type polyedges yield $E \geq n - 1$, and the aforementioned relation yields un upper bound of $p \leq 4n - 2(n - 1)$:

$$P_{h\times w,p}(n) = 0 \quad \text{if} \quad p > 2n + 2. \tag{8}$$

Given these lower and upper bounds, the following summation is well defined:

$$\sum_{p} P_{w\times h,p}(n) = P_{w\times h}(n). \tag{9}$$

1.2. **Read's Results.** For $n \leq 10$ and $w \neq h$ these numbers have been tabulated by Read [12]. Note that his tables $b(q, n)$ for width 2, $c(q, n)$ for width 3 and $d(q, n)$ for width 4 are not reducing the fixed polyominoes to free polyominoes if width and height are the same [2, A259088], because he only applied a symmetry group of order 4 in all cases. One needs to compare our results with his $z_w(n)$ in cases where $w = h$.

As observed by Klarner [5], the actual discrepancy with his data is for $P_{5\times 5}(10) = 529$ where Read reports only $z_5(10) = 340$, such that our total is $P(10) = 4655$, not his 4466.

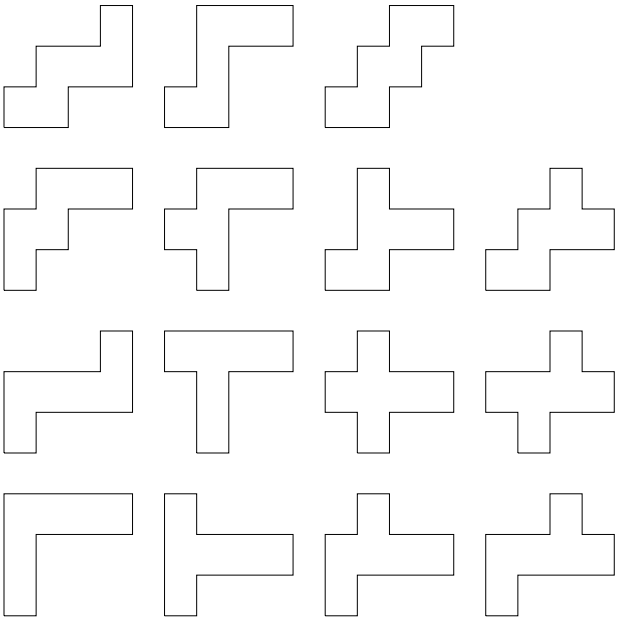

FIGURE 1. The $P_{4\times3}(6) = 15$ polyominoes of size 6 with a 4×3 bounding box.

1.3. **Kurz's Hulls of Maximum Area.** The following data are not compatible with Kurz's count of n-ominoes with maximum area hw inside the bounding box [7]. If for example $n = 6$, the maximum area $hw = 12$ is achieved with $h = 4$, $w = 3$, and we generate the $P_{4\times3}(6) = 15$ free polyominoes in Figure 1, whereas his formula gives only $(n^3 - 2n^2 + 4n + 8)/16 = 11$.

2. Algorithm

The construction of free n-ominoes by the program in the Appendix includes the following steps:

2.1. **Compositions Look-up Tables.** Each polyomino is represented by a $h \times w$ matrix of zeros and ones, where zeros represent unoccupied and ones occupied squares, respectively. For a given n and a given height $h \le n$ of the bounding box, the matrix row sums are a rough classification of the n-ominoes. A list of the compositions of n into h parts of minimum size 1 and maximum size w is compiled, where the i-th part is the sum of the ith row of the binary matrix. The lower limit is 1 because the connectivity of the polyominoes requires at least one occupied square in each row, and the upper limit equals the width. Because free polyominoes are unchanged by flipping the rows along the middle axis, compositions are discarded which are lexicographically larger than their reversed associates. For each of these row sums (from 1 up to w) we also keep a list of the $2^w - 1$ possible bitsets (compositions into w parts that are either 0 or 1), representing a single row of at least 1 and at most w 1's in the binary matrix.

2.2. **Row-by-row Stacking.** In a outer loop over the compositions, the first row of the binary matrix is any of the bitsets (inner loop) compatible with the first part of the composition. The other rows are recursively filled with bitsets with a sum equal to the associated part of the composition, and requiring that at least one of the squares of the row has a common edge with a square of the previous row to ensure that adjacent rows are edge-connected. [The bit-wise AND-operation between adjacent rows must not be zero.] This is similar to printing an object layer-by-layer in stereolithography [4, 14].

2.3. **Reduction by Symmetry.** As the last row of the matrix has been filled with a bitset, we have essentially constructed a candidate of a fixed n-omino. The program checks first that the set of squares is connected, because that is not guaranteed by the stacking method. [The growth may have lead to separated columns.] In a loop over the 4 (or 8) symmetry operations of flips and rotations, a lexicographically smallest representative of the polyomino is selected, a free polyomino. This is compared with the members in the set of free $h \times w$ n-polyominoes constructed so far, and added to the set if it is "new."

3. Results

The numerical results are summarized in the following table. Each entry has one of these four formats:

- Four positive integer numbers n, h, w and p, a colon and $P_{h\times w,p}(n)$. These are the size of the free n-omino, the height and width of the bounding box, the perimeter, and the number of free n-ominoes of perimeter p fitting in that bounding box.
- Three positive integer numbers n, h and w, a colon and $P_{h\times w}(n)$. These are the size of the free n-omino, the height and width of the bounding box, and the number of free n-ominoes fitting in that bounding box. The 6-ominoes of Figure 1 are shown as `6 4 3 : 15`, for example.
- Two positive integer numbers n and p, a bar and $\sum_{w,h} P_{h\times w,p}(n)$. These are the size of the free n-omino, the perimeter, and the number of free n-ominoes with that perimeter. These agree with Brendan Owens tables [2, A342243].
- One positive integer number n, a colon and $P(n)$. This is the size of the n-omino and the number of free n-ominoes of that size. This entry is absent for $n \geq 18$ because some of the $P_{h\times w}(n)$ have not yet been computed. For all cases computed, $P(n)$ matches the results in the literature [2, A000105][10], which shows the integrity of the program.

For $n \leq 14$ results agree with the published table [11]; for $n = 15$ they correct their numbers, and for $n > 15$ they seem to be new.

```
1 1 1 4 : 1        1 1 1 : 1          1 4 | 1            1  :  1

2 2 1 6 : 1        2 2 1 : 1          2 6 | 1            2  :  1

3 3 1 8 : 1        3 3 1 : 1          3 2 2 8 : 1        3 2 2 : 1
3 8 | 2            3  :  2

4 4 1 10 : 1       4 4 1 : 1          4 2 2 8 : 1        4 2 2 : 1
```

```
4 3 2 10 : 3        4 3 2 : 3          4 8 | 1            4 10 | 4
4  :  5

5 5 1 12 : 1        5 5 1 : 1          5 3 2 10 : 1       5 3 2 12 : 1
5 3 2 : 2           5 4 2 12 : 3       5 4 2 : 3          5 3 3 12 : 6
5 3 3 : 6           5 10 | 1           5 12 | 11          5  :  12

6 6 1 14 : 1        6 6 1 : 1          6 3 2 10 : 1       6 3 2 : 1
6 4 2 12 : 3        6 4 2 14 : 3       6 4 2 : 6          6 5 2 14 : 5
6 5 2 : 5           6 3 3 12 : 4       6 3 3 14 : 3       6 3 3 : 7
6 4 3 14 : 15       6 4 3 : 15         6 10 | 1           6 12 | 7
6 14 | 27          6  :  35

7 7 1 16 : 1        7 7 1 : 1          7 4 2 12 : 1       7 4 2 14 : 1
7 4 2 : 2           7 5 2 14 : 3       7 5 2 16 : 8       7 5 2 : 11
7 6 2 16 : 5        7 6 2 : 5          7 3 3 12 : 3       7 3 3 14 : 1
7 3 3 16 : 3        7 3 3 : 7          7 4 3 14 : 16      7 4 3 16 : 23
7 4 3 : 39          7 5 3 16 : 25      7 5 3 : 25         7 4 4 16 : 18
7 4 4 : 18          7 12 | 4           7 14 | 21          7 16 | 83
7  :  108

8 8 1 18 : 1        8 8 1 : 1          8 4 2 12 : 1       8 4 2 : 1
8 5 2 14 : 3        8 5 2 16 : 5       8 5 2 18 : 2       8 5 2 : 10
8 6 2 16 : 5        8 6 2 18 : 14      8 6 2 : 19         8 7 2 18 : 7
8 7 2 : 7           8 3 3 12 : 1       8 3 3 14 : 1       8 3 3 16 : 1
8 3 3 : 3           8 4 3 14 : 17      8 4 3 16 : 20      8 4 3 18 : 22
8 4 3 : 59          8 5 3 16 : 30      8 5 3 18 : 66      8 5 3 : 96
8 6 3 18 : 35       8 6 3 : 35         8 4 4 16 : 30      8 4 4 18 : 47
8 4 4 : 77          8 5 4 18 : 61      8 5 4 : 61         8 12 | 2
8 14 | 21          8 16 | 91          8 18 | 255         8  :  369

9 9 1 20 : 1        9 9 1 : 1          9 5 2 14 : 1       9 5 2 16 : 2
9 5 2 : 3           9 6 2 16 : 3       9 6 2 18 : 13      9 6 2 20 : 6
9 6 2 : 22          9 7 2 18 : 5       9 7 2 20 : 23      9 7 2 : 28
9 8 2 20 : 7        9 8 2 : 7          9 3 3 12 : 1       9 3 3 : 1
9 4 3 14 : 8        9 4 3 16 : 13      9 4 3 18 : 13      9 4 3 20 : 8
9 4 3 : 42          9 5 3 16 : 36      9 5 3 18 : 84      9 5 3 20 : 90
9 5 3 : 210         9 6 3 18 : 48      9 6 3 20 : 140     9 6 3 : 188
9 7 3 20 : 49       9 7 3 : 49         9 4 4 16 : 35      9 4 4 18 : 62
9 4 4 20 : 84       9 4 4 : 181        9 5 4 18 : 114     9 5 4 20 : 269
9 5 4 : 383         9 6 4 20 : 97      9 6 4 : 97         9 5 5 20 : 73
9 5 5 : 73          9 12 | 1           9 14 | 9           9 16 | 89
9 18 | 339          9 20 | 847         9  :  1285

10 10 1 22 : 1      10 10 1 : 1        10 5 2 14 : 1      10 5 2 : 1
10 6 2 16 : 3       10 6 2 18 : 8      10 6 2 20 : 4      10 6 2 : 15
10 7 2 18 : 5       10 7 2 20 : 25     10 7 2 22 : 22     10 7 2 : 52
10 8 2 20 : 7       10 8 2 22 : 33     10 8 2 : 40        10 9 2 22 : 9
```

```
10 9 2 : 9          10 4 3 14 : 5       10 4 3 16 : 5       10 4 3 18 : 8
10 4 3 20 : 3       10 4 3 : 21         10 5 3 16 : 26      10 5 3 18 : 75
10 5 3 20 : 92      10 5 3 22 : 62      10 5 3 : 255        10 6 3 18 : 61
10 6 3 20 : 232     10 6 3 22 : 257     10 6 3 : 550        10 7 3 20 : 70
10 7 3 22 : 262     10 7 3 : 332        10 8 3 22 : 63      10 8 3 : 63
10 4 4 16 : 33      10 4 4 18 : 63      10 4 4 20 : 97      10 4 4 22 : 73
10 4 4 : 266        10 5 4 18 : 173     10 5 4 20 : 471     10 5 4 22 : 660
10 5 4 : 1304       10 6 4 20 : 211     10 6 4 22 : 611     10 6 4 : 822
10 7 4 22 : 155     10 7 4 : 155        10 5 5 20 : 148     10 5 5 22 : 381
10 5 5 : 529        10 6 5 22 : 240     10 6 5 : 240        10 14 | 6
10 16 | 67         10 18 | 393         10 20 | 1360        10 22 | 2829
10  :  4655
```

```
11 11 1 24 : 1      11 11 1 : 1         11 6 2 16 : 1       11 6 2 18 : 2
11 6 2 : 3          11 7 2 18 : 3       11 7 2 20 : 21      11 7 2 22 : 18
11 7 2 24 : 3       11 7 2 : 45         11 8 2 20 : 5       11 8 2 22 : 41
11 8 2 24 : 44      11 8 2 : 90         11 9 2 22 : 7       11 9 2 24 : 46
11 9 2 : 53         11 10 2 24 : 9      11 10 2 : 9         11 4 3 14 : 1
11 4 3 16 : 2       11 4 3 18 : 1       11 4 3 : 4          11 5 3 16 : 20
11 5 3 18 : 41      11 5 3 20 : 79      11 5 3 22 : 44      11 5 3 24 : 28
11 5 3 : 212        11 6 3 18 : 54      11 6 3 20 : 249     11 6 3 22 : 376
11 6 3 24 : 275     11 6 3 : 954        11 7 3 20 : 92      11 7 3 22 : 489
11 7 3 24 : 650     11 7 3 : 1231       11 8 3 22 : 96      11 8 3 24 : 433
11 8 3 : 529        11 9 3 24 : 81      11 9 3 : 81         11 4 4 16 : 22
11 4 4 18 : 42      11 4 4 20 : 79      11 4 4 22 : 71      11 4 4 24 : 37
11 4 4 : 251        11 5 4 18 : 182     11 5 4 20 : 602     11 5 4 22 : 1069
11 5 4 24 : 994     11 5 4 : 2847       11 6 4 20 : 330     11 6 4 22 : 1301
11 6 4 24 : 1917    11 6 4 : 3548       11 7 4 22 : 340     11 7 4 24 : 1211
11 7 4 : 1551       11 8 4 24 : 220     11 8 4 : 220        11 5 5 20 : 256
11 5 5 22 : 815     11 5 5 24 : 1342    11 5 5 : 2413       11 6 5 22 : 588
11 6 5 24 : 1778    11 6 5 : 2366       11 7 5 24 : 410     11 7 5 : 410
11 6 6 24 : 255     11 6 6 : 255        11 14 | 1          11 16 | 45
11 18 | 325        11 20 | 1713        11 22 | 5255       11 24 | 9734
11  :  17073
```

```
12 12 1 26 : 1      12 12 1 : 1         12 6 2 16 : 1       12 6 2 : 1
12 7 2 18 : 3       12 7 2 20 : 10      12 7 2 22 : 8       12 7 2 : 21
12 8 2 20 : 5       12 8 2 22 : 39      12 8 2 24 : 60      12 8 2 26 : 15
12 8 2 : 119        12 9 2 22 : 7       12 9 2 24 : 61      12 9 2 26 : 90
12 9 2 : 158        12 10 2 24 : 9      12 10 2 26 : 60     12 10 2 : 69
12 11 2 26 : 11     12 11 2 : 11        12 4 3 14 : 1       12 4 3 : 1
12 5 3 16 : 8       12 5 3 18 : 24      12 5 3 20 : 31      12 5 3 22 : 28
12 5 3 24 : 11      12 5 3 26 : 1       12 5 3 : 103        12 6 3 18 : 48
12 6 3 20 : 190     12 6 3 22 : 405     12 6 3 24 : 342     12 6 3 26 : 199
12 6 3 : 1184       12 7 3 20 : 90      12 7 3 22 : 618     12 7 3 24 : 1168
12 7 3 26 : 924     12 7 3 : 2800       12 8 3 22 : 129     12 8 3 24 : 892
12 8 3 26 : 1385    12 8 3 : 2406       12 9 3 24 : 126     12 9 3 26 : 674
12 9 3 : 800        12 10 3 26 : 99     12 10 3 : 99        12 4 4 16 : 14
```

```
12 4 4 18 : 25     12 4 4 20 : 44     12 4 4 22 : 46     12 4 4 24 : 35
12 4 4 26 : 4      12 4 4 : 168       12 5 4 18 : 175    12 5 4 20 : 547
12 5 4 22 : 1254   12 5 4 24 : 1466   12 5 4 26 : 999    12 5 4 : 4441
12 6 4 20 : 413    12 6 4 22 : 1936   12 6 4 24 : 3963   12 6 4 26 : 4011
12 6 4 : 10323     12 7 4 22 : 572    12 7 4 24 : 2910   12 7 4 26 : 4757
12 7 4 : 8239      12 8 4 24 : 520    12 8 4 26 : 2160   12 8 4 : 2680
12 9 4 26 : 313    12 9 4 : 313       12 5 5 20 : 325    12 5 5 22 : 1277
12 5 5 24 : 2772   12 5 5 26 : 3001   12 5 5 : 7375      12 6 5 22 : 1093
12 6 5 24 : 4418   12 6 5 26 : 7650   12 6 5 : 13161     12 7 5 24 : 1070
12 7 5 26 : 3668   12 7 5 : 4738      12 8 5 26 : 646    12 8 5 : 646
12 6 6 24 : 687    12 6 6 26 : 2148   12 6 6 : 2835      12 7 6 26 : 908
12 7 6 : 908       12 14 | 1          12 16 | 23        12 18 | 275
12 20 | 1655      12 22 | 7412       12 24 | 20510     12 26 | 33724
12  :  63600
```

```
13 13 1 28 : 1     13 13 1 : 1        13 7 2 18 : 1      13 7 2 20 : 3
13 7 2 : 4         13 8 2 20 : 3      13 8 2 22 : 26     13 8 2 24 : 36
13 8 2 26 : 8      13 8 2 : 73        13 9 2 22 : 5      13 9 2 24 : 64
13 9 2 26 : 132    13 9 2 28 : 56     13 9 2 : 257       13 10 2 24 : 7
13 10 2 26 : 85    13 10 2 28 : 146   13 10 2 : 238      13 11 2 26 : 9
13 11 2 28 : 77    13 11 2 : 86       13 12 2 28 : 11    13 12 2 : 11
13 5 3 16 : 5      13 5 3 18 : 7      13 5 3 20 : 14     13 5 3 22 : 6
13 5 3 24 : 1      13 5 3 : 33        13 6 3 18 : 29     13 6 3 20 : 125
13 6 3 22 : 268    13 6 3 24 : 295    13 6 3 26 : 182    13 6 3 28 : 65
13 6 3 : 964       13 7 3 20 : 90     13 7 3 22 : 567    13 7 3 24 : 1450
13 7 3 26 : 1559   13 7 3 28 : 968    13 7 3 : 4634      13 8 3 22 : 134
13 8 3 24 : 1226   13 8 3 26 : 2932   13 8 3 28 : 2526   13 8 3 : 6818
13 9 3 24 : 172    13 9 3 26 : 1462   13 9 3 28 : 2679   13 9 3 : 4313
13 10 3 26 : 160   13 10 3 28 : 982   13 10 3 : 1142     13 11 3 28 : 121
13 11 3 : 121      13 4 4 16 : 6      13 4 4 18 : 10     13 4 4 20 : 19
13 4 4 22 : 19     13 4 4 24 : 11     13 4 4 26 : 1      13 4 4 : 66
13 5 4 18 : 127    13 5 4 20 : 441    13 5 4 22 : 984    13 5 4 24 : 1515
13 5 4 26 : 1331   13 5 4 28 : 610    13 5 4 : 5008      13 6 4 20 : 439
13 6 4 22 : 2121   13 6 4 24 : 5671   13 6 4 26 : 7767   13 6 4 28 : 5997
13 6 4 : 21995     13 7 4 22 : 743    13 7 4 24 : 4829   13 7 4 26 : 11431
13 7 4 28 : 12439  13 7 4 : 29442     13 8 4 24 : 883    13 8 4 26 : 5658
13 8 4 28 : 10280  13 8 4 : 16821     13 9 4 26 : 742    13 9 4 28 : 3585
13 9 4 : 4327      13 10 4 28 : 415   13 10 4 : 415      13 5 5 20 : 374
13 5 5 22 : 1493   13 5 5 24 : 4079   13 5 5 26 : 6077   13 5 5 28 : 5018
13 5 5 : 17041     13 6 5 22 : 1547   13 6 5 24 : 7817   13 6 5 26 : 19041
13 6 5 28 : 22728  13 6 5 : 51133     13 7 5 24 : 2083   13 7 5 26 : 10150
13 7 5 28 : 18765  13 7 5 : 30998     13 8 5 26 : 1790   13 8 5 28 : 6893
13 8 5 : 8683      13 9 5 28 : 979    13 9 5 : 979       13 6 6 24 : 1366
13 6 6 26 : 6008   13 6 6 28 : 11159  13 6 6 : 18533     13 7 6 26 : 2710
13 7 6 28 : 9242   13 7 6 : 11952     13 8 6 28 : 1553   13 8 6 : 1553
13 7 7 28 : 950    13 7 7 : 950       13 16 | 11        13 18 | 174
13 20 | 1508      13 22 | 7913       13 24 | 31505     13 26 | 79235
13 28 | 118245    13  :  238591
```

```
14 14 1 30 : 1     14 14 1 : 1        14 7 2 18 : 1      14 7 2 : 1
14 8 2 20 : 3      14 8 2 22 : 13     14 8 2 24 : 12     14 8 2 : 28
14 9 2 22 : 5      14 9 2 24 : 50     14 9 2 26 : 120    14 9 2 28 : 56
14 9 2 30 : 6      14 9 2 : 237       14 10 2 24 : 7     14 10 2 26 : 94
14 10 2 28 : 260   14 10 2 30 : 144   14 10 2 : 505      14 11 2 26 : 9
14 11 2 28 : 113   14 11 2 30 : 238   14 11 2 : 360      14 12 2 28 : 11
14 12 2 30 : 95    14 12 2 : 106      14 13 2 30 : 13    14 13 2 : 13
14 5 3 16 : 1      14 5 3 18 : 3      14 5 3 20 : 2      14 5 3 : 6
14 6 3 18 : 20     14 6 3 20 : 68     14 6 3 22 : 152    14 6 3 24 : 150
14 6 3 26 : 112    14 6 3 28 : 41     14 6 3 30 : 3      14 6 3 : 546
14 7 3 20 : 68     14 7 3 22 : 449    14 7 3 24 : 1253   14 7 3 26 : 1758
14 7 3 28 : 1359   14 7 3 30 : 610    14 7 3 : 5497      14 8 3 22 : 140
14 8 3 24 : 1303   14 8 3 26 : 4070   14 8 3 28 : 5292   14 8 3 30 : 3377
14 8 3 : 14182     14 9 3 24 : 186    14 9 3 26 : 2153   14 9 3 28 : 6296
14 9 3 30 : 6087   14 9 3 : 14722     14 10 3 26 : 221   14 10 3 28 : 2240
14 10 3 30 : 4710  14 10 3 : 7171     14 11 3 28 : 198   14 11 3 30 : 1382
14 11 3 : 1580     14 12 3 30 : 143   14 12 3 : 143      14 4 4 16 : 3
14 4 4 18 : 4      14 4 4 20 : 8      14 4 4 22 : 4      14 4 4 24 : 1
14 4 4 : 20        14 5 4 18 : 91     14 5 4 20 : 278    14 5 4 22 : 683
14 5 4 24 : 1054   14 5 4 26 : 1150   14 5 4 28 : 708    14 5 4 30 : 204
14 5 4 : 4168      14 6 4 20 : 412    14 6 4 22 : 2042   14 6 4 24 : 5820
14 6 4 26 : 10406  14 6 4 28 : 10952  14 6 4 30 : 6403   14 6 4 : 36035
14 7 4 22 : 888    14 7 4 24 : 6023   14 7 4 26 : 18783  14 7 4 28 : 29226
14 7 4 30 : 24235  14 7 4 : 79155     14 8 4 24 : 1220   14 8 4 26 : 10098
14 8 4 28 : 27956  14 8 4 30 : 32468  14 8 4 : 71742     14 9 4 26 : 1307
14 9 4 28 : 9969   14 9 4 30 : 20300  14 9 4 : 31576     14 10 4 28 : 1029
14 10 4 30 : 5605  14 10 4 : 6634     14 11 4 30 : 551   14 11 4 : 551
14 5 5 20 : 356    14 5 5 22 : 1528   14 5 5 24 : 4355   14 5 5 26 : 8433
14 5 5 28 : 9696   14 5 5 30 : 5952   14 5 5 : 30320     14 6 5 22 : 1962
14 6 5 24 : 10565  14 6 5 26 : 32900  14 6 5 28 : 56245  14 6 5 30 : 51450
14 6 5 : 153122    14 7 5 24 : 3131   14 7 5 26 : 19614  14 7 5 28 : 52979
14 7 5 30 : 67506  14 7 5 : 143230    14 8 5 26 : 3597   14 8 5 28 : 20703
14 8 5 30 : 40936  14 8 5 : 65236     14 9 5 28 : 2814   14 9 5 30 : 12080
14 9 5 : 14894     14 10 5 30 : 1415  14 10 5 : 1415     14 6 6 24 : 2134
14 6 6 26 : 11776  14 6 6 28 : 31832  14 6 6 30 : 41232  14 6 6 : 86974
14 7 6 26 : 5753   14 7 6 28 : 28395  14 7 6 30 : 55064  14 7 6 : 89212
14 8 6 28 : 4978   14 8 6 30 : 18237  14 8 6 : 23215     14 9 6 30 : 2555
14 9 6 : 2555      14 7 7 28 : 3005   14 7 7 30 : 10397  14 7 7 : 13402
14 8 7 30 : 3417   14 8 7 : 3417      14 16 | 4          14 18 | 119
14 20 | 1195      14 22 | 7866       14 24 | 37264     14 26 | 132354
14 28 | 306353    14 30 | 416816     14  :  901971

15 15 1 32 : 1     15 15 1 : 1        15 8 2 20 : 1      15 8 2 22 : 3
15 8 2 : 4         15 9 2 22 : 3      15 9 2 24 : 34     15 9 2 26 : 60
15 9 2 28 : 22     15 9 2 : 119       15 10 2 24 : 5     15 10 2 26 : 82
15 10 2 28 : 264   15 10 2 30 : 208   15 10 2 32 : 32    15 10 2 : 591
15 11 2 26 : 7     15 11 2 28 : 131   15 11 2 30 : 438   15 11 2 32 : 319
```

15 11 2 : 895 15 12 2 28 : 9 15 12 2 30 : 145 15 12 2 32 : 344
15 12 2 : 498 15 13 2 30 : 11 15 13 2 32 : 116 15 13 2 : 127
15 14 2 32 : 13 15 14 2 : 13 15 5 3 16 : 1 15 5 3 : 1
15 6 3 18 : 8 15 6 3 20 : 32 15 6 3 22 : 53 15 6 3 24 : 58
15 6 3 26 : 31 15 6 3 28 : 5 15 6 3 : 187 15 7 3 20 : 52
15 7 3 22 : 295 15 7 3 24 : 939 15 7 3 26 : 1286 15 7 3 28 : 1307
15 7 3 30 : 651 15 7 3 32 : 215 15 7 3 : 4745 15 8 3 22 : 122
15 8 3 24 : 1172 15 8 3 26 : 4115 15 8 3 28 : 7074 15 8 3 30 : 6367
15 8 3 32 : 3161 15 8 3 : 22011 15 9 3 24 : 202 15 9 3 26 : 2482
15 9 3 28 : 9594 15 9 3 30 : 14653 15 9 3 32 : 9989 15 9 3 : 36920
15 10 3 26 : 246 15 10 3 28 : 3435 15 10 3 30 : 12028 15 10 3 32 : 13053
15 10 3 : 28762 15 11 3 28 : 276 15 11 3 30 : 3243 15 11 3 32 : 7785
15 11 3 : 11304 15 12 3 30 : 240 15 12 3 32 : 1867 15 12 3 : 2107
15 13 3 32 : 169 15 13 3 : 169 15 4 4 16 : 1 15 4 4 18 : 1
15 4 4 20 : 1 15 4 4 : 3 15 5 4 18 : 48 15 5 4 20 : 161
15 5 4 22 : 361 15 5 4 24 : 599 15 5 4 26 : 640 15 5 4 28 : 467
15 5 4 30 : 149 15 5 4 32 : 14 15 5 4 : 2439 15 6 4 20 : 336
15 6 4 22 : 1656 15 6 4 24 : 5072 15 6 4 26 : 9854 15 6 4 28 : 13248
15 6 4 30 : 10896 15 6 4 32 : 4686 15 6 4 : 45748 15 7 4 22 : 910
15 7 4 24 : 6533 15 7 4 26 : 22741 15 7 4 28 : 46214 15 7 4 30 : 54977
15 7 4 32 : 35979 15 7 4 : 167354 15 8 4 24 : 1503 15 8 4 26 : 13818
15 8 4 28 : 50377 15 8 4 30 : 88344 15 8 4 32 : 76956 15 8 4 : 230998
15 9 4 26 : 1832 15 9 4 28 : 18740 15 9 4 30 : 60151 15 9 4 32 : 75284
15 9 4 : 156007 15 10 4 28 : 1820 15 10 4 30 : 16331 15 10 4 32 : 36933
15 10 4 : 55084 15 11 4 30 : 1368 15 11 4 32 : 8383 15 11 4 : 9751
15 12 4 32 : 698 15 12 4 : 698 15 5 5 20 : 319 15 5 5 22 : 1305
15 5 5 24 : 4092 15 5 5 26 : 8312 15 5 5 28 : 12496 15 5 5 30 : 10969
15 5 5 32 : 5177 15 5 5 : 42670 15 6 5 22 : 2159 15 6 5 24 : 12519
15 6 5 26 : 42263 15 6 5 28 : 94480 15 6 5 30 : 125012 15 6 5 32 : 90399
15 6 5 : 366832 15 7 5 24 : 4227 15 7 5 26 : 29165 15 7 5 28 : 102315
15 7 5 30 : 192889 15 7 5 32 : 188706 15 7 5 : 517302 15 8 5 26 : 5638
15 8 5 28 : 42638 15 8 5 30 : 127763 15 8 5 32 : 172051 15 8 5 : 348090
15 9 5 28 : 5797 15 9 5 30 : 38616 15 9 5 32 : 82083 15 9 5 : 126496
15 10 5 30 : 4216 15 10 5 32 : 19998 15 10 5 : 24214 15 11 5 32 : 1991
15 11 5 : 1991 15 6 6 24 : 2918 15 6 6 26 : 17904 15 6 6 28 : 62144
15 6 6 30 : 119623 15 6 6 32 : 120742 15 6 6 : 323331 15 7 6 26 : 9499
15 7 6 28 : 60082 15 7 6 30 : 174572 15 7 6 32 : 239176 15 7 6 : 483329
15 8 6 28 : 10956 15 8 6 30 : 60506 15 8 6 32 : 122449 15 8 6 : 193911
15 9 6 30 : 8498 15 9 6 32 : 33656 15 9 6 : 42154 15 10 6 32 : 3965
15 10 6 : 3965 15 7 7 28 : 6804 15 7 7 30 : 34813 15 7 7 32 : 69679
15 7 7 : 111296 15 8 7 30 : 11926 15 8 7 32 : 43057 15 8 7 : 54983
15 9 7 32 : 6003 15 9 7 : 6003 15 8 8 32 : 3473 15 8 8 : 3473
15 16 | 2 15 18 | 57 15 20 | 902 15 22 | 6867
15 24 | 39873 15 26 | 169975 15 28 | 550695 15 30 | 1179603
15 32 | 1478602 15 : 3426576

16 16 1 34 : 1 16 16 1 : 1 16 8 2 20 : 1 16 8 2 : 1
16 9 2 22 : 3 16 9 2 24 : 15 16 9 2 26 : 18 16 9 2 : 36

16 10 2 24 : 5 16 10 2 26 : 64 16 10 2 28 : 196 16 10 2 30 : 142
16 10 2 32 : 22 16 10 2 : 429 16 11 2 26 : 7 16 11 2 28 : 122
16 11 2 30 : 520 16 11 2 32 : 560 16 11 2 34 : 144 16 11 2 : 1353
16 12 2 28 : 9 16 12 2 30 : 173 16 12 2 32 : 700 16 12 2 34 : 611
16 12 2 : 1493 16 13 2 30 : 11 16 13 2 32 : 181 16 13 2 34 : 498
16 13 2 : 690 16 14 2 32 : 13 16 14 2 34 : 138 16 14 2 : 151
16 15 2 34 : 15 16 15 2 : 15 16 6 3 18 : 5 16 6 3 20 : 10
16 6 3 22 : 19 16 6 3 24 : 11 16 6 3 26 : 2 16 6 3 : 47
16 7 3 20 : 29 16 7 3 22 : 188 16 7 3 24 : 514 16 7 3 26 : 785
16 7 3 28 : 717 16 7 3 30 : 440 16 7 3 32 : 146 16 7 3 34 : 14
16 7 3 : 2833 16 8 3 22 : 106 16 8 3 24 : 906 16 8 3 26 : 3604
16 8 3 28 : 6643 16 8 3 30 : 7849 16 8 3 32 : 5045 16 8 3 34 : 1944
16 8 3 : 26097 16 9 3 24 : 188 16 9 3 26 : 2496 16 9 3 28 : 10863
16 9 3 30 : 22262 16 9 3 32 : 22861 16 9 3 34 : 12090 16 9 3 : 70760
16 10 3 26 : 272 16 10 3 28 : 4224 16 10 3 30 : 19675 16 10 3 32 : 35121
16 10 3 34 : 25633 16 10 3 : 84925 16 11 3 28 : 314 16 11 3 30 : 5160
16 11 3 32 : 21060 16 11 3 34 : 25711 16 11 3 : 52245 16 12 3 30 : 337
16 12 3 32 : 4516 16 12 3 34 : 12144 16 12 3 : 16997 16 13 3 32 : 286
16 13 3 34 : 2466 16 13 3 : 2752 16 14 3 34 : 195 16 14 3 : 195
16 4 4 16 : 1 16 4 4 : 1 16 5 4 18 : 27 16 5 4 20 : 75
16 5 4 22 : 173 16 5 4 24 : 256 16 5 4 26 : 281 16 5 4 28 : 155
16 5 4 30 : 39 16 5 4 32 : 2 16 5 4 : 1008 16 6 4 20 : 253
16 6 4 22 : 1228 16 6 4 24 : 3682 16 6 4 26 : 7843 16 6 4 28 : 11371
16 6 4 30 : 11584 16 6 4 32 : 7215 16 6 4 34 : 2113 16 6 4 : 45289
16 7 4 22 : 878 16 7 4 24 : 6149 16 7 4 26 : 23535 16 7 4 28 : 53349
16 7 4 30 : 82564 16 7 4 32 : 78204 16 7 4 34 : 40696 16 7 4 : 285375
16 8 4 24 : 1688 16 8 4 26 : 16393 16 8 4 28 : 68514 16 8 4 30 : 157693
16 8 4 32 : 206510 16 8 4 34 : 143690 16 8 4 : 594488 16 9 4 26 : 2361
16 9 4 28 : 27330 16 9 4 30 : 116817 16 9 4 32 : 228778 16 9 4 34 : 210019
16 9 4 : 585305 16 10 4 28 : 2635 16 10 4 30 : 31888 16 10 4 32 : 11749
16 10 4 34 : 15887 16 10 4 : 310890 16 11 4 30 : 2474 16 11 4 32 : 25296
16 11 4 34 : 63357 16 11 4 : 91127 16 12 4 32 : 1786 16 12 4 34 : 12066
16 12 4 : 13852 16 13 4 34 : 885 16 13 4 : 885 16 5 5 20 : 242
16 5 5 22 : 1028 16 5 5 24 : 3168 16 5 5 26 : 7039 16 5 5 28 : 11270
16 5 5 30 : 12941 16 5 5 32 : 8657 16 5 5 34 : 2999 16 5 5 : 47344
16 6 5 22 : 2236 16 6 5 24 : 12806 16 6 5 26 : 47425 16 6 5 28 : 116197
16 6 5 30 : 202288 16 6 5 32 : 215151 16 6 5 34 : 124141 16 6 5 : 720244
16 7 5 24 : 5041 16 7 5 26 : 37943 16 7 5 28 : 149350 16 7 5 30 : 368821
16 7 5 32 : 538996 16 7 5 34 : 424291 16 7 5 : 1524442 16 8 5 26 : 7907
16 8 5 28 : 68003 16 8 5 30 : 267620 16 8 5 32 : 550460 16 8 5 34 : 565173
16 8 5 : 1459163 16 9 5 28 : 9362 16 9 5 30 : 83452 16 9 5 32 : 277096
16 9 5 34 : 393768 16 9 5 : 763678 16 10 5 30 : 8853 16 10 5 32 : 67190
16 10 5 34 : 15353 16 10 5 : 229573 16 11 5 32 : 6078 16 11 5 34 : 31630
16 11 5 : 37708 16 12 5 34 : 2715 16 12 5 : 2715 16 6 6 24 : 3582
16 6 6 26 : 23923 16 6 6 28 : 92067 16 6 6 30 : 233150 16 6 6 32 : 351491
16 6 6 34 : 287447 16 6 6 : 991660 16 7 6 26 : 13814 16 7 6 28 : 98837
16 7 6 30 : 373501 16 7 6 32 : 778788 16 7 6 34 : 832594 16 7 6 : 2097534
16 8 6 28 : 18964 16 8 6 30 : 135909 16 8 6 32 : 422862 16 8 6 34 : 604444

16 8 6 : 1182179 16 9 6 30 : 19394 16 9 6 32 : 118891 16 9 6 34 : 251944
16 9 6 : 390229 16 10 6 32 : 13783 16 10 6 34 : 58782 16 10 6 : 72565
16 11 6 34 : 5985 16 11 6 : 5985 16 7 7 28 : 11927 16 7 7 30 : 78956
16 7 7 32 : 241244 16 7 7 34 : 343854 16 7 7 : 675981 16 8 7 30 : 28311
16 8 7 32 : 155127 16 8 7 34 : 317389 16 8 7 : 500827 16 9 7 32 : 22078
16 9 7 34 : 83522 16 9 7 : 105600 16 10 7 34 : 10001 16 10 7 : 10001
16 8 8 32 : 12931 16 8 8 34 : 46797 16 8 8 : 59728 16 9 8 34 : 12859
16 9 8 : 12859 16 16 | 1 16 18 | 32 16 20 | 610
16 22 | 5859 16 24 | 38011 16 26 | 195712 16 28 | 762419
16 30 | 2272824 16 32 | 4536616 16 34 | 5267171 16 : 13079255

17 17 1 36 : 1 17 17 1 : 1 17 9 2 22 : 1 17 9 2 24 : 4
17 9 2 : 5 17 10 2 24 : 3 17 10 2 26 : 39 17 10 2 28 : 90
17 10 2 30 : 40 17 10 2 : 172 17 11 2 26 : 5 17 11 2 28 : 105
17 11 2 30 : 440 17 11 2 32 : 528 17 11 2 34 : 160 17 11 2 36 : 10
17 11 2 : 1248 17 12 2 28 : 7 17 12 2 30 : 170 17 12 2 32 : 876
17 12 2 34 : 1240 17 12 2 36 : 416 17 12 2 : 2709 17 13 2 30 : 9
17 13 2 32 : 222 17 13 2 34 : 1032 17 13 2 36 : 1080 17 13 2 : 2343
17 14 2 32 : 11 17 14 2 34 : 221 17 14 2 36 : 670 17 14 2 : 902
17 15 2 34 : 13 17 15 2 36 : 163 17 15 2 : 176 17 16 2 36 : 15
17 16 2 : 15 17 6 3 18 : 1 17 6 3 20 : 3 17 6 3 22 : 2
17 6 3 : 6 17 7 3 20 : 20 17 7 3 22 : 92 17 7 3 24 : 255
17 7 3 26 : 320 17 7 3 28 : 301 17 7 3 30 : 150 17 7 3 32 : 34
17 7 3 34 : 1 17 7 3 : 1173 17 8 3 22 : 72 17 8 3 24 : 644
17 8 3 26 : 2578 17 8 3 28 : 5103 17 8 3 30 : 6363 17 8 3 32 : 5104
17 8 3 34 : 2406 17 8 3 36 : 613 17 8 3 : 22883 17 9 3 24 : 177
17 9 3 26 : 2162 17 9 3 28 : 10590 17 9 3 30 : 24536 17 9 3 32 : 33258
17 9 3 34 : 25099 17 9 3 36 : 10668 17 9 3 : 106490 17 10 3 26 : 266
17 10 3 28 : 4542 17 10 3 30 : 24363 17 10 3 32 : 58458 17 10 3 34 : 68067
17 10 3 36 : 37973 17 10 3 : 193669 17 11 3 28 : 354 17 11 3 30 : 6605
17 11 3 32 : 36422 17 11 3 34 : 74989 17 11 3 36 : 59516 17 11 3 : 177886
17 12 3 30 : 390 17 12 3 32 : 7356 17 12 3 34 : 34464 17 12 3 36 : 46936
17 12 3 : 89146 17 13 3 32 : 404 17 13 3 34 : 6072 17 13 3 36 : 18184
17 13 3 : 24660 17 14 3 34 : 336 17 14 3 36 : 3168 17 14 3 : 3504
17 15 3 36 : 225 17 15 3 : 225 17 5 4 18 : 10 17 5 4 20 : 31
17 5 4 22 : 61 17 5 4 24 : 87 17 5 4 26 : 65 17 5 4 28 : 16
17 5 4 30 : 1 17 5 4 : 271 17 6 4 20 : 164 17 6 4 22 : 781
17 6 4 24 : 2372 17 6 4 26 : 5087 17 6 4 28 : 7855 17 6 4 30 : 8491
17 6 4 32 : 6246 17 6 4 34 : 2712 17 6 4 36 : 404 17 6 4 : 34112
17 7 4 22 : 740 17 7 4 24 : 5347 17 7 4 26 : 20547 17 7 4 28 : 52199
17 7 4 30 : 89474 17 7 4 32 : 109466 17 7 4 34 : 83681 17 7 4 36 : 33884
17 7 4 : 395338 17 8 4 24 : 1741 17 8 4 26 : 17059 17 8 4 28 : 79526
17 8 4 30 : 208830 17 8 4 32 : 359646 17 8 4 34 : 375423 17 8 4 36 : 214398
17 8 4 : 1256623 17 9 4 26 : 2733 17 9 4 28 : 34700 17 9 4 30 : 172190
17 9 4 32 : 448262 17 9 4 34 : 639620 17 9 4 36 : 467195 17 9 4 : 1764700
17 10 4 28 : 3439 17 10 4 30 : 48703 17 10 4 32 : 24126 17 10 4 34 : 52698
17 10 4 36 : 51061 17 10 4 : 1331013 17 11 4 30 : 3609 17 11 4 32 : 50863
17 11 4 34 : 21232 17 11 4 36 : 31114 17 11 4 : 577936 17 12 4 32 : 3237

```
17 12 4 34 : 37480 17 12 4 36 : 10303 17 12 4 : 143749   17 13 4 34 : 2266
17 13 4 36 : 16853 17 13 4 : 19119    17 14 4 36 : 1085   17 14 4 : 1085
17 5 5 20 : 177    17 5 5 22 : 703    17 5 5 24 : 2222    17 5 5 26 : 4898
17 5 5 28 : 8581   17 5 5 30 : 10362  17 5 5 32 : 9021    17 5 5 34 : 4275
17 5 5 36 : 1091   17 5 5 : 41330     17 6 5 22 : 2064    17 6 5 24 : 12068
17 6 5 26 : 45444  17 6 5 28 : 122676 17 6 5 30 : 237638 17 6 5 32 : 330961
17 6 5 34 : 286249 17 6 5 36 : 131634 17 6 5 : 1168734   17 7 5 24 : 5677
17 7 5 26 : 43420  17 7 5 28 : 188768 17 7 5 30 : 527045 17 7 5 32 : 101356
17 7 5 34 : 120633 17 7 5 36 : 781238 17 7 5 : 3766042   17 8 5 26 : 9928
17 8 5 28 : 94486  17 8 5 30 : 428091 17 8 5 32 : 115814 17 8 5 34 : 183065
17 8 5 36 : 151804 17 8 5 : 5039358   17 9 5 28 : 13505   17 9 5 30 : 140130
17 9 5 32 : 618315 17 9 5 34 : 137888 17 9 5 36 : 147982 17 9 5 : 3630653
17 10 5 30 : 14637 17 10 5 32 : 15070 17 10 5 34 : 55274 17 10 5 36 : 82931
17 10 5 : 1547398  17 11 5 32 : 12967 17 11 5 34 : 11058 17 11 5 36 : 27164
17 11 5 : 395198   17 12 5 34 : 8490  17 12 5 36 : 48133 17 12 5 : 56623
17 13 5 36 : 3630  17 13 5 : 3630     17 6 6 24 : 4090    17 6 6 26 : 28061
17 6 6 28 : 119313 17 6 6 30 : 338527 17 6 6 32 : 677250 17 6 6 34 : 836520
17 6 6 36 : 564067 17 6 6 : 2567828   17 7 6 26 : 17922   17 7 6 28 : 142390
17 7 6 30 : 611085 17 7 6 32 : 168259 17 7 6 34 : 275779 17 7 6 36 : 241832
17 7 6 : 7630113   17 8 6 28 : 28589  17 8 6 30 : 237447 17 8 6 32 : 969941
17 8 6 34 : 215926 17 8 6 36 : 240434 17 8 6 : 5799584   17 9 6 30 : 34575
17 9 6 32 : 280200 17 9 6 34 : 933135 17 9 6 36 : 138598 17 9 6 : 2633896
17 10 6 32 : 32196 17 10 6 34 : 21876 17 10 6 36 : 48623 17 10 6 : 737198
17 11 6 34 : 21348 17 11 6 36 : 98080 17 11 6 : 119428   17 12 6 36 : 8689
17 12 6 : 8689     17 7 7 28 : 18328  17 7 7 30 : 139288 17 7 7 32 : 557378
17 7 7 34 : 123363 17 7 7 36 : 138548 17 7 7 : 3334120   17 8 7 30 : 52255
17 8 7 32 : 372496 17 8 7 34 : 118795 17 8 7 36 : 174339 17 8 7 : 3356103
17 9 7 32 : 54476  17 9 7 34 : 320296 17 9 7 36 : 672895 17 9 7 : 1047667
17 10 7 34 : 38500 17 10 7 36 : 15347 17 10 7 : 191974   17 11 7 36 : 16025
17 11 7 : 16025    17 8 8 32 : 32399  17 8 8 34 : 181055 17 8 8 36 : 373895
17 8 8 : 587349    17 9 8 34 : 51366  17 9 8 36 : 190611 17 9 8 : 241977
17 10 8 36 : 22827 17 10 8 : 22827    17 9 9 36 : 13006  17 9 9 : 13006
17 18 | 11         17 20 | 395        17 22 | 4516       17 24 | 34687
17 26 | 200534     17 28 | 935463     17 30 | 3365444    17 32 | 9314277
17 34 | 17412438   17 36 | 18840144   17 : 50107909

18 18 1 38 : 1     18 18 1 : 1        18 9 2 22 : 1      18 9 2 : 1
18 10 2 24 : 3     18 10 2 26 : 18    18 10 2 28 : 24    18 10 2 : 45
18 11 2 26 : 5     18 11 2 28 : 75    18 11 2 30 : 294   18 11 2 32 : 280
18 11 2 34 : 66    18 11 2 : 720      18 12 2 28 : 7     18 12 2 30 : 155
18 12 2 32 : 860   18 12 2 34 : 1408  18 12 2 36 : 688   18 12 2 38 : 74
18 12 2 : 3192     18 13 2 30 : 9     18 13 2 32 : 226   18 13 2 34 : 1400
18 13 2 36 : 2408  18 13 2 38 : 1054  18 13 2 : 5097     18 14 2 32 : 11
18 14 2 34 : 276   18 14 2 36 : 1476  18 14 2 38 : 1768  18 14 2 : 3531
18 15 2 34 : 13    18 15 2 36 : 265   18 15 2 38 : 902   18 15 2 : 1180
18 16 2 36 : 15    18 16 2 38 : 189   18 16 2 : 204      18 17 2 38 : 17
18 17 2 : 17       18 6 3 18 : 1      18 6 3 : 1         18 7 3 20 : 8
18 7 3 22 : 43     18 7 3 24 : 83     18 7 3 26 : 105    18 7 3 28 : 68
```

18 7 3 30 : 16 18 7 3 32 : 1 18 7 3 : 324 18 8 3 22 : 52
18 8 3 24 : 416 18 8 3 26 : 1637 18 8 3 28 : 3100 18 8 3 30 : 4033
18 8 3 32 : 3220 18 8 3 34 : 1725 18 8 3 36 : 528 18 8 3 38 : 50
18 8 3 : 14761 18 9 3 24 : 140 18 9 3 26 : 1720 18 9 3 28 : 8825
18 9 3 30 : 22375 18 9 3 32 : 33828 18 9 3 34 : 32710 18 9 3 36 : 18786
18 9 3 38 : 6129 18 9 3 : 124513 18 10 3 26 : 262 18 10 3 28 : 4337
18 10 3 30 : 25808 18 10 3 32 : 72320 18 10 3 34 : 11196 18 10 3 36 : 95443
18 10 3 38 : 42898 18 10 3 : 353037 18 11 3 28 : 356 18 11 3 30 : 7495
18 11 3 32 : 48257 18 11 3 34 : 13458 18 11 3 36 : 17625 18 11 3 38 : 10432
18 11 3 : 471268 18 12 3 30 : 444 18 12 3 32 : 9753 18 12 3 34 : 62228
18 12 3 36 : 14630 18 12 3 38 : 12643 18 12 3 : 345168 18 13 3 32 : 474
18 13 3 34 : 10119 18 13 3 36 : 53476 18 13 3 38 : 80836 18 13 3 : 144905
18 14 3 34 : 477 18 14 3 36 : 7960 18 14 3 38 : 26207 18 14 3 : 34644
18 15 3 36 : 390 18 15 3 38 : 4006 18 15 3 : 4396 18 16 3 38 : 255
18 16 3 : 255 18 5 4 18 : 5 18 5 4 20 : 10 18 5 4 22 : 21
18 5 4 24 : 16 18 5 4 26 : 3 18 5 4 : 55 18 6 4 20 : 102
18 6 4 22 : 454 18 6 4 24 : 1342 18 6 4 26 : 2854 18 6 4 28 : 4363
18 6 4 30 : 4696 18 6 4 32 : 3592 18 6 4 34 : 1571 18 6 4 36 : 294
18 6 4 38 : 14 18 6 4 : 19282 18 7 4 22 : 602 18 7 4 24 : 4143
18 7 4 26 : 16308 18 7 4 28 : 42850 18 7 4 30 : 80905 18 7 4 32 : 109660
18 7 4 34 : 106315 18 7 4 36 : 64374 18 7 4 38 : 19119 18 7 4 : 444276
18 8 4 24 : 1667 18 8 4 26 : 16429 18 8 4 28 : 79762 18 8 4 30 : 234276
18 8 4 32 : 460211 18 8 4 34 : 629410 18 8 4 36 : 541781 18 8 4 38 : 254168
18 8 4 : 2217704 18 9 4 26 : 3019 18 9 4 28 : 38943 18 9 4 30 : 217453
18 9 4 32 : 658018 18 9 4 34 : 124522 18 9 4 36 : 140821 18 9 4 38 : 851089
18 9 4 : 4421955 18 10 4 28 : 4136 18 10 4 30 : 64934 18 10 4 32 : 37924
18 10 4 34 : 11102 18 10 4 36 : 17236 18 10 4 38 : 13148 18 10 4 : 4597056
18 11 4 30 : 4826 18 11 4 32 : 80401 18 11 4 34 : 45637 18 11 4 36 : 11063
18 11 4 38 : 11367 18 11 4 : 2784608 18 12 4 32 : 4818 18 12 4 34 : 77114
18 12 4 36 : 36089 18 12 4 38 : 57222 18 12 4 : 1015049 18 13 4 34 : 4169
18 13 4 36 : 53563 18 13 4 38 : 16087 18 13 4 : 218608 18 14 4 36 : 2839
18 14 4 38 : 22919 18 14 4 : 25758 18 15 4 38 : 1331 18 15 4 : 1331
18 5 5 20 : 107 18 5 5 22 : 441 18 5 5 24 : 1348 18 5 5 26 : 3046
18 5 5 28 : 5193 18 5 5 30 : 6760 18 5 5 32 : 6135 18 5 5 34 : 3497
18 5 5 36 : 1104 18 5 5 38 : 133 18 5 5 : 27764 18 6 5 22 : 1797
18 6 5 24 : 10260 18 6 5 26 : 39758 18 6 5 28 : 110426 18 6 5 30 : 235718
18 6 5 32 : 367374 18 6 5 34 : 414656 18 6 5 36 : 288850 18 6 5 38 : 105468
18 6 5 : 1574307 18 7 5 24 : 5858 18 7 5 26 : 45971 18 7 5 28 : 207375
18 7 5 30 : 644811 18 7 5 32 : 141247 18 7 5 34 : 221214 18 7 5 36 : 219737
18 7 5 38 : 117728 18 7 5 : 7903297 18 8 5 26 : 11772 18 8 5 28 : 116548
18 8 5 30 : 590362 18 8 5 32 : 184123 18 8 5 34 : 384501 18 8 5 36 : 494357
18 8 5 38 : 341922 18 8 5 : 14767740 18 9 5 28 : 17531 18 9 5 30 : 204650
18 9 5 32 : 105835 18 9 5 34 : 312982 18 9 5 36 : 529706 18 9 5 38 : 457715
18 9 5 : 14284582 18 10 5 30 : 21559 18 10 5 32 : 26325 18 10 5 34 : 12971
18 10 5 36 : 31279 18 10 5 38 : 35043 18 10 5 : 8214249 18 11 5 32 : 21837
18 11 5 34 : 25554 18 11 5 36 : 10303 18 11 5 38 : 16340 18 11 5 : 2941738
18 12 5 34 : 18349 18 12 5 36 : 17393 18 12 5 38 : 45826 18 12 5 : 650544
18 13 5 36 : 11550 18 13 5 38 : 70926 18 13 5 : 82476 18 14 5 38 : 4746

```
18 14 5 : 4746     18 18 | 6          18 20 | 227        18 22 | 3411

19 19 1 40 : 1     19 19 1 : 1        19 10 2 24 : 1     19 10 2 26 : 4
19 10 2 : 5        19 11 2 26 : 3     19 11 2 28 : 47    19 11 2 30 : 126
19 11 2 32 : 73    19 11 2 : 249      19 12 2 28 : 5     19 12 2 30 : 123
19 12 2 32 : 660   19 12 2 34 : 1040  19 12 2 36 : 480   19 12 2 38 : 48
19 12 2 : 2356     19 13 2 30 : 7     19 13 2 32 : 216   19 13 2 34 : 1460
19 13 2 36 : 3118  19 13 2 38 : 2080  19 13 2 40 : 354   19 13 2 : 7235
19 14 2 32 : 9     19 14 2 34 : 290   19 14 2 36 : 2064  19 14 2 38 : 4256
19 14 2 40 : 2240  19 14 2 : 8859     19 15 2 34 : 11    19 15 2 36 : 337
19 15 2 38 : 2010  19 15 2 40 : 2755  19 15 2 : 5113     19 16 2 36 : 13
19 16 2 38 : 313   19 16 2 40 : 1156  19 16 2 : 1482     19 17 2 38 : 15
19 17 2 40 : 218   19 17 2 : 233      19 18 2 40 : 17    19 18 2 : 17
19 7 3 20 : 5      19 7 3 22 : 12     19 7 3 24 : 27     19 7 3 26 : 16
19 7 3 28 : 4      19 7 3 : 64        19 8 3 22 : 29     19 8 3 24 : 243
19 8 3 26 : 831    19 8 3 28 : 1567   19 8 3 30 : 1806   19 8 3 32 : 1409
19 8 3 34 : 680    19 8 3 36 : 155    19 8 3 38 : 10     19 8 3 : 6730
19 9 3 24 : 111    19 9 3 26 : 1241   19 9 3 28 : 6663   19 9 3 30 : 16828
19 9 3 32 : 27757  19 9 3 34 : 28527  19 9 3 36 : 20484  19 9 3 38 : 8528
19 9 3 40 : 1972   19 9 3 : 112111    19 10 3 26 : 228   19 10 3 28 : 3763
19 10 3 30 : 23774 19 10 3 32 : 74451 19 10 3 34 : 13269 19 10 3 36 : 14759
19 10 3 38 : 96934 19 10 3 40 : 35236 19 10 3 : 514669   19 11 3 28 : 361
19 11 3 30 : 7612  19 11 3 32 : 54822 19 11 3 34 : 18097 19 11 3 36 : 31912
19 11 3 38 : 30178 19 11 3 40 : 14311 19 11 3 : 1007794  19 12 3 30 : 458
19 12 3 32 : 11464 19 12 3 34 : 86931 19 12 3 36 : 27852 19 12 3 38 : 40866
19 12 3 40 : 25761 19 12 3 : 1043651  19 13 3 32 : 546   19 13 3 34 : 13736
19 13 3 36 : 99982 19 13 3 38 : 26513 19 13 3 40 : 25024 19 13 3 : 629639
19 14 3 34 : 566   19 14 3 36 : 13469 19 14 3 38 : 79472 19 14 3 40 : 13221
19 14 3 : 225722   19 15 3 36 : 556   19 15 3 38 : 10189 19 15 3 40 : 36700
19 15 3 : 47445    19 16 3 38 : 448   19 16 3 40 : 4965  19 16 3 : 5413
19 17 3 40 : 289   19 17 3 : 289

20 20 1 42 : 1     20 20 1 : 1        20 10 2 24 : 1     20 10 2 : 1
20 11 2 26 : 3     20 11 2 28 : 20    20 11 2 30 : 32    20 11 2 : 55
20 12 2 28 : 5     20 12 2 30 : 89    20 12 2 32 : 408   20 12 2 34 : 493
20 12 2 36 : 146   20 12 2 : 1141     20 13 2 30 : 7     20 13 2 32 : 183
20 13 2 34 : 1290  20 13 2 36 : 2800  20 13 2 38 : 2064  20 13 2 40 : 432
20 13 2 42 : 20    20 13 2 : 6796     20 14 2 32 : 9     20 14 2 34 : 286
20 14 2 36 : 2324  20 14 2 38 : 6038  20 14 2 40 : 5198  20 14 2 42 : 1186
20 14 2 : 15041    20 15 2 34 : 11    20 15 2 36 : 362   20 15 2 38 : 2952
20 15 2 40 : 7008  20 15 2 42 : 4400  20 15 2 : 14733    20 16 2 36 : 13
20 16 2 38 : 403   20 16 2 40 : 2684  20 16 2 42 : 4095  20 16 2 : 7195
20 17 2 38 : 15    20 17 2 40 : 365   20 17 2 42 : 1482  20 17 2 : 1862
20 18 2 40 : 17    20 18 2 42 : 248   20 18 2 : 265      20 19 2 42 : 19
20 19 2 : 19       20 7 3 20 : 1      20 7 3 22 : 4      20 7 3 24 : 3
20 7 3 : 8         20 8 3 22 : 20     20 8 3 24 : 119    20 8 3 26 : 378
20 8 3 28 : 577    20 8 3 30 : 616    20 8 3 32 : 378    20 8 3 34 : 110
20 8 3 36 : 9      20 8 3 : 2207      20 9 3 24 : 72     20 9 3 26 : 864
```

```
20 9 3 28 : 4308   20 9 3 30 : 11049  20 9 3 32 : 17691  20 9 3 34 : 19274
20 9 3 36 : 13919  20 9 3 38 : 6741   20 9 3 40 : 1875   20 9 3 42 : 189
20 9 3 : 75982     20 10 3 26 : 197   20 10 3 28 : 2994  20 10 3 30 : 20067
20 10 3 32 : 65088 20 10 3 34 : 12842 20 10 3 36 : 15975 20 10 3 38 : 13456
20 10 3 40 : 68226 20 10 3 42 : 19559 20 10 3 : 598863   20 11 3 28 : 332
20 11 3 30 : 7148  20 11 3 32 : 54526 20 11 3 34 : 20406 20 11 3 36 : 42304
20 11 3 38 : 53011 20 11 3 40 : 38709 20 11 3 42 : 15041 20 11 3 : 1756745
20 12 3 30 : 474   20 12 3 32 : 12232 20 12 3 34 : 10437 20 12 3 36 : 40071
20 12 3 38 : 79815 20 12 3 40 : 83265 20 12 3 42 : 41514 20 12 3 : 2563739
20 13 3 32 : 572   20 13 3 34 : 16646 20 13 3 36 : 14554 20 13 3 38 : 53017
20 13 3 40 : 86636 20 13 3 42 : 58451 20 13 3 : 2143813  20 14 3 34 : 656
20 14 3 36 : 18690 20 14 3 38 : 15281 20 14 3 40 : 45271 20 14 3 42 : 46528
20 14 3 : 1090162  20 15 3 36 : 666   20 15 3 38 : 17510 20 15 3 40 : 11399
20 15 3 42 : 20754 20 15 3 : 339710   20 16 3 38 : 641   20 16 3 40 : 12812
20 16 3 42 : 50027 20 16 3 : 63480    20 17 3 40 : 510   20 17 3 42 : 6082
20 17 3 : 6592     20 18 3 42 : 323   20 18 3 : 323

21 21 1 44 : 1     21 21 1 : 1        21 11 2 26 : 1     21 11 2 28 : 5
21 11 2 : 6        21 12 2 28 : 3     21 12 2 30 : 52    21 12 2 32 : 168
21 12 2 34 : 112   21 12 2 : 335      21 13 2 30 : 5     21 13 2 32 : 146
21 13 2 34 : 924   21 13 2 36 : 1832  21 13 2 38 : 1120  21 13 2 40 : 174
21 13 2 : 4201     21 14 2 32 : 7     21 14 2 34 : 255   21 14 2 36 : 2190
21 14 2 38 : 6200  21 14 2 40 : 6240  21 14 2 42 : 2064  21 14 2 44 : 160
21 14 2 : 17116    21 15 2 34 : 9     21 15 2 36 : 367   21 15 2 38 : 3440
21 15 2 40 : 10672 21 15 2 42 : 11200 21 15 2 44 : 3276  21 15 2 : 28964
21 16 2 36 : 11    21 16 2 38 : 442   21 16 2 40 : 4020  21 16 2 42 : 10920
21 16 2 44 : 7872  21 16 2 : 23265    21 17 2 38 : 13    21 17 2 40 : 476
21 17 2 42 : 3468  21 17 2 44 : 5888  21 17 2 : 9845    21 18 2 40 : 15
21 18 2 42 : 421   21 18 2 44 : 1834  21 18 2 : 2270    21 19 2 42 : 17
21 19 2 44 : 281   21 19 2 : 298      21 20 2 44 : 19    21 20 2 : 19
21 7 3 20 : 1      21 7 3 : 1         21 8 3 22 : 8      21 8 3 24 : 51
21 8 3 26 : 117    21 8 3 28 : 164    21 8 3 30 : 118    21 8 3 32 : 32
21 8 3 34 : 2      21 8 3 : 492       21 9 3 24 : 52     21 9 3 26 : 529
21 9 3 28 : 2543   21 9 3 30 : 5948   21 9 3 32 : 9339   21 9 3 34 : 9437
21 9 3 36 : 6655   21 9 3 38 : 3004   21 9 3 40 : 735    21 9 3 42 : 62
21 9 3 44 : 2      21 9 3 : 38306     21 10 3 26 : 145   21 10 3 28 : 2257
21 10 3 30 : 15147 21 10 3 32 : 50177 21 10 3 34 : 10198 21 10 3 36 : 13698
21 10 3 38 : 12575 21 10 3 40 : 78869 21 10 3 42 : 30256 21 10 3 44 : 6060
21 10 3 : 547633   21 11 3 28 : 307   21 11 3 30 : 6138  21 11 3 32 : 49742
21 11 3 34 : 19850 21 11 3 36 : 46109 21 11 3 38 : 66741 21 11 3 40 : 63498
21 11 3 42 : 36693 21 11 3 44 : 11735 21 11 3 : 2502466

22 22 1 : 1        22 11 2 : 1        22 12 2 : 66       22 13 2 : 1725
22 14 2 : 13353    22 15 2 : 39392    22 16 2 : 52556    22 17 2 : 35573
22 18 2 : 13189    22 19 2 : 2768     22 20 2 : 334      22 21 2 : 21
```

For fixed n and w, the sums $\sum_{h=w}^{\lfloor n/w \rfloor} P_{h\times w}(n)$ are summarized in Table 1. A closed-form formula for the values 1, 1, 6, 18, 73, 255,... on its diagonal is known

$n\backslash w$	1	2	3	4	5	6	7	8	9	$P(n)$
1	1									1
2	1									1
3	1	1								2
4	1	4								5
5	1	5	6							12
6	1	12	22							35
7	1	18	71	18						108
8	1	37	193	138						369
9	1	60	490	661	73					1285
10	1	117	1221	2547	769					4655
11	1	200	3011	8417	5189	255				17073
12	1	379	7393	26164	25920	3743				63600
13	1	669	18025	78074	108834	32038	950			238591
14	1	1250	43847	229881	408217	201956	16819			901971
15	1	2247	106206	668082	1427595	1046690	172282	3473		3426576
16	1	4168	256851	1928220	4784867	4740152	1292409	72587		13079255
17	1	7570	619642	5523946	15648966	19496736	7945889	852153	13006	50107909
18	1	13987	1493272	15745682	50451443					192622052
19	1	25549	3593527							
20	1	47108	8641624							
21	1	86319								
22	1	158978								

TABLE 1. The number of free n-ominoes with a bounding box of short edge w.

[2, A057051][6]. An equivalent triangle for fixed n-ominoes is also in the OEIS [2, A308359].

The sums $\sum_{w\ge 1}^{n} P_{w\times w}(n)$ for the free polynomials with a square bounding box have their their own OEIS entry [2, A259088].

For fixed w and h, the sums $\sum_{n=w+h-1}^{wh} P_{h\times w}(n)$ are collected in [2, A268371], in particular for $w = 2$ in [2, A335711].

APPENDIX A. THE JAVA PROGRAM

A.1. **Compilation and Use.** The Java classes that follow are compiled as usual with

```
javac de/mpg/mpia/rjm/{Composit.java,FreePoly.java,FreePolySet.java,FreePolySetThrd.java}
jar cfe FreePolySet.jar de.mpg.mpia.rjm.FreePolySet de/mpg/mpia/rjm/*
```

The compiled source code is also available in https://www.mpia.de/~mathar/progs/FreePolySet.jar. The main program is then called with

```
java -cp . de.mpg.mpia.rjm.FreePolySet [-v] [-j #] [-f] [-w #] [-h #] [-s {N,R,H,V,HVR}] n
```

respectively

```
java -jar FreePolySet.jar [-v] [-j #] [-f] [-w #] [-h #] [-s {N,R,H,V,HVR}] n
```

where the last argument is the size (number of cells) of the polyomino, a positive integer.

The option `-v` lets the program print the (0,1)-matrix for each polyomino that is constructed. This is done in two formats: (i) a list of row and column indices of the occupied cells in parenthesis `(`r,c`)` where rows and columns count from 0 upwards. (ii) a sequence of zeros and ones in the shape of the bounding box, where the ones are the occupied and the zeros the empty cells in the square lattice. Note that the lines with the counts of Section 3 (recognized by containing colons) are printed anyway.

The option `-f` lets the program handle *fixed* polyominoes without reduction for the symmetry groups [2, A001168,A292357,A308359].

The option `-j` followed by a positive integer number uses parallel threads (as many as given by the followup integer) to construct the n-ominoes of a given width and height. If the option is not used, only a single thread is run.

The option `-w` followed by a positive integer number lets the program consider only a bounding box of a specific width; this will only generate any output if the width is in the range from 1 to n. If the option is not used, the program will execute a sequential loop over all widths which are commensurate with n.

The option `-h` followed by a positive integer number lets the program consider only a bounding box of a specific height. If the option is not used, the program will execute a sequential loop over all heights in the range w to $n - w + 1$.

The options `-w` and `-h` support parallelization of the computations on the operating system level by calling it more than once at the same time for different widths and/or heights.

The option `-s` followed by a string with a subset of Redelmeier's capital letters [13] filters the polyominoes with respect to symmetry. The letter N constructs only polyominoes without symmetry, the letter R constructs only those with a symmetry of a 180° rotation about the center of the bounding box, the letter H constructs only those with a symmetry of flipping along the short axis, the letter V constructs only those with a symmetry of flipping along the long axis. [The short axis of the bounding box has length w, the long axis length h.] The string HVR selects polyominoes that have all of the symmetries H, V and R.

The call

```
java -jar FreePolySet.jar -v 5
```

for example would show the 12 free pentominoes.

A.2. **Classes.** The class `Composit` constructs compositions of some number n into k parts given lower and upper bounds for the size of each part. This is done by standard recursion and filling the vector of parts left-to-right.

The class `FreePoly` represents a binary matrix of zeros and ones with given height (number of rows) and width (number of columns). It has member functions that rotate and/or flip the cells and a member function to select from these variants one normalized view of the free n-omino.

The class `FreePolySet` is the main program which first selects the size n of the polyominoes and the height and width of the bounding box, runs the double loop over compositions of n into h parts and bitsets with w parts (which amounts essentially to explicit construction of roughly a quarter of all fixed n-ominoes), and copies the free polyomino representatives into a set of eventually $P(n)$ binary matrices.

Appendix B. Source code of de/mpg/mpia/rjm/Composit.java

```
/*
* $Header: de/mpg/mpia/rjm/Composit.java$
*/

/** @file
* A class which generates the compositions of an integer into a fixed number of parts.
* @author R. J. Mathar
* @see <a href="http://vixra.org/abs/1905.0474">vixra:1905.0474</a>
*/

package de.mpg.mpia.rjm ;

import java.util.* ;
import java.lang.* ;

/**
* @brief The set of compositions of some fixed positive integer.
* @since 2019-05-11
* @author Richard J. Matar
*/
public class Composit
{
    /** the sum of the parts
    */
    int n ;

    /** the number of parts
    */
    int k ;

    /** the lowest size a part may have
    */
    int minPart ;

    /** the largest size a part may have
    */
    int maxPart ;

    /** The compositions to be generated.
    * Each composition is represented as a 1-dimensionsal array
    * of k numbers in the range minPart..maxPart and sum n.
    */
    Vector<int[]> comps;

    /**
    * Constructor defining the integer to be partitioned.
    * This is not just defining the task at hand but actually generating
    * all compositions within the ctor.
    * @param n The sum of the parts
    * @param k The number of the parts
    * @param minP The smallest size any part may have.
    * @param maxP The largest size any part may have.
    * @since 2019-05-11
    */
    public Composit(int n, int k, int minP, int maxP)
    {
        this.n = n ;
        this.k = k ;
        minPart = minP ;
        maxPart = maxP ;
        /* the initially empty set of compositions.
        */
        comps= new Vector<int[]>() ;

        /* Generate the vector comps[] if basic requirements are met.
        *  Each part is >= minPart, so the total is >=k*minPart.
        *  Each part is <= maxPart, so the total is <=k*maxPart.
        */
        if ( k*minPart <= n && k*maxPart >= n)
            comps = generate( new int[0],n) ;
```

```
} /* ctor */

/**
* @return The number of compositions.
* Because the compositions are all generated with the ctor,
*  this number is available right away.
*/
public int size()
{
    return comps.size() ;
} /* size */

/** generated recursively the compositions of n
* @param given The initial sublist of parts already fixed.
* @param nResid The sum over the elements not yet in given[].
* @return The partitions represented as vectors of length k.
*/
private Vector<int[]> generate(int[] given, int nResid)
{
    /* the compositions that can be generated;
    * The result of this subroutine
    */
    Vector<int[]> subcomp = new Vector<int[]>() ;

    if ( nResid < 0 || given.length > k)
    {
        /* prefixed parts not compatible with requirements;
        * return with the empty set.
        */
        return subcomp;
    }

    if ( given.length == k)
    {
        if ( nResid  ==0 )
            subcomp.add(given.clone()) ;
        /* Return a vector of 0 or 1 elements composing n.
        */
        return subcomp;
    }

    /* here given.length < k and nResid >=0
    */
    if ( given.length == k-1)
    {
        /* one final part to  be appended to the given[].
        * need a part of the size nResid to fill up to
        * to n
        */
        if (nResid >= minPart && nResid <= maxPart )
        {
            int[] c = new int[k] ;
            for(int pi =0 ; pi < c.length ; pi++)
                c[pi] = (pi < given.length) ? given[pi] : nResid ;
            subcomp.add(c) ;
        }
    }
    else
    {
        int[] c = new int[given.length+1] ;
        for(int pi=0 ; pi < given.length ; pi++)
            c[pi] = given[pi] ;

        /* 2 or more parts to  be appended; number of missing
        * parts is k-given.length, each part >=minPart. Let p be the
        * part to be appended next. The minimum total of the unassigned
        * parts  is p+minPart*(k-given.length-1). This value must stay <= nResid.
        * p <= nResid -minPart*(k-given.length-1).
        * The maximum total of the unassigned
        * parts  is p+maxPart*(k-given.length-1); this value must stay >=nResid
        * p >= nResid-maxPart*(k-given.length-1).
        */
```

```
            final int nextmin = Math.max(minPart, nResid-maxPart*(k-given.length-1)) ;
            final int nextmax = Math.min(maxPart, nResid-minPart*(k-given.length-1)) ;
            for(int p = nextmin ; p <= nextmax ; p++)
            {
                /* fill in the last integer into the parts list */
                c[given.length] = p ;
                final Vector<int[]> iters = generate(c,nResid-p) ;
                subcomp.addAll(iters) ;
            }
        }

        return subcomp ;
    } /* generate */

    /** Compare two integer vectors element-wise left to right.
    * If the two vectors have different length, the longer one is considered larger.
    * If the two vectors have the same length, the lexicographic comparison
    *  (comparing elements at index 0, 1, 2..) is executed. The vector
    *  which first has a larger element than the other is the larger vector.
    * @return -1, 0 or +1 if left is considered smaller than, equal to or larger than right.
    */
    public static int compareTo(final int[] left, final int[] right)
    {
        if ( left.length > right.length)
            return 1;
        else if ( left.length < right.length)
            return -1;
        else
        {
            for(int i=0 ; i < left.length ; i++)
            {
                if ( left[i] > right[i])
                    return 1;
                else if ( left[i] < right[i])
                    return -1 ;
            }
            return 0 ;
        }

    } /* compareTo */


    /** Reverse the integers in a vector
    * @param arg The initial vector.
    * @param return The initial vector where arg[i] has been swapped with arg[length-1-i].
    */
    public static int[] reverse(final int[] arg)
    {
        int[] rev =new int[arg.length] ;
        for(int i=0 ; i < arg.length ; i++)
            rev[i] = arg[arg.length-1-i] ;
        return rev ;
    } /* reverse */

    /** List all compositions
    * @return The compositions [c00,c01...],[c10,c11...]
    */
    public String toString()
    {
        String str = new String();
        for (int[] p : comps)
        {
            str += "[" ;
            for (int pi = 0 ; pi < p.length ; pi++)
                str += p[pi] +"," ;
            str += "]" ;
        }
        return str ;
    } /* toString */

    /** Test program
    */
```

```
    public static void main(String[] args)
    {
        for(int n = 0 ; n < 6 ; n++)
        for(int k = 0 ; k < 6 ; k++)
        {
            Composit c = new Composit(n,k,1,n) ;
            System.out.println("n "+n + " k " +k + " : " + c.comps.size() + " " + c.toString()) ;
        }
    } /* main */
} /* class Composit */
```

Appendix C. Source code of de/mpg/mpia/rjm/FreePoly.java

```
/*
* $Header: de/mpg/mpia/rjm/FreePoly.java$
*/

/** @file
* A n-omino with a r times c bounding box.
* @author R. J. Mathar
* @see <a href="http://vixra.org/abs/1905.0474">vixra:1905.0474</a>
*/

package de.mpg.mpia.rjm ;

import java.util.* ;
import java.lang.* ;

/**
* @brief A free n-omino with a tight bound box (convex rectangular hull) of r rows and c columns.
* @since 2019-05-11
* @author Richard J. Mathar
*/
public class FreePoly
{
    /** the sum of the parts
    */
    int n ;

    /** the number of rows
    */
    int rows ;

    /** the number of columns
    */
    int cols ;

    /** The array of zeros and ones for each cell indexed by row and column
    */
    byte[][] bits ;


    /**
    * Constructor with a predefined n-omino.
    * @param zeroone The array of the zeros and ones.
    * @param freep If true, construct free polynomios.
    *  That means store a representation considered the same if rotated/flipped.
    * @param Read is a flag: if zero or positive, exclude 90degree rotation symmetries.
    *  That means if negative, the equivalence operations of reducing
    *  the set of occupied cells to a single representative are all tested,
    *  of which there are 4 operations (group of identity, two flips and 180 deg rotation)
    *  for rectangular shapes, and 8 operations (those by adding 90 degree rotations)
    *  for the square shapes. If the Read parameter is zero or positive, that additional
    *  set of 4 operations (90 degree rotations) is *not* added to the square shapes,
    *  and all results will be blind to those symmetries, i.e., this is not what
    *  the usual meaning for free polyominoes would do.
    * @since 2019-05-11
    */
    public FreePoly(final byte[][] zeroone, boolean freep, int Read)
    {
        rows = zeroone.length ;
        if ( rows > 0 )
```

```
        cols = zeroone[0].length ;
    else
        cols = 0 ;
    bits = new byte[rows][cols] ;
    n=0 ;
    /* clone the elements of zeroone (which may be modified later
    * by the calling program)
    */
    for (int r=0 ; r < rows ; r++)
    for (int c=0 ; c < cols ; c++)
    {
        bits[r][c] = zeroone[r][c] ;
        n += bits[r][c] ;
    }

    if (freep)
        reduce(Read) ;
} /* ctor */

/**
* Constructor with a predefined n-omino.
* @param zeroone The array of the zeros and ones.
* @param freep If true, construct free polynomios.
*  That means store a representation counted as one if rotated/flipped.
* @param Read is a flag: if zero or positive, exclude 90degree rotation symmetries.
*  That means if negative, the equivalence operations of reducing
*  the set of occupied cells to a single representative are all tested,
*  of which there are 4 operations (group of identity, two flips and 180 deg rotation)
*  for rectangular shapes, and 8 operations (those by adding 90 degree rotations)
*  for the square shapes. If the Read parameter is zero or positive, that additional
*  set of 4 operations (90 degree rotations) is *not* added to the square shapes,
*  and all results will be blind to those symmetries, i.e., this is not what
*  the usual meaning for free polyominoes would do.
* @since 2019-05-11
*/
public FreePoly(final int[][] zeroone, boolean freep, int Read)
{
    this(zeroone, zeroone.length, freep, Read) ;
} /* ctor */

/**
* Constructor with a predefined n-omino.
* @param zeroone The array of the zeros and ones.
* @param rowsToKeep The number of valid rows in zeroone.
*   By default this should be the same as zeroone.length, but the
*   number can be chosen to be less such that the exceeding rows in zeroone will
*   be ignored.
* @param freep If true, construct free polynomios.
*  That means store a representation counted as one if rotated/flipped.
* @param Read is a flag: if zero or positive, exclude 90degree rotation symmetries.
*  That means if negative, the equivalence operations of reducing
*  the set of occupied cells to a single representative are all tested,
*  of which there are 4 operations (group of identity, two flips and 180 deg rotation)
*  for rectangular shapes, and 8 operations (those by adding 90 degree rotations)
*  for the square shapes. If the Read parameter is zero or positive, that additional
*  set of 4 operations (90 degree rotations) is *not* added to the square shapes,
*  and all results will be blind to those symmetries, i.e., this is not what
*  the usual meaning for free polyominoes would do.
* @since 2020-06-16
*/
public FreePoly(final int[][] zeroone, int rowsToKeep, boolean freep, int Read)
{
    if ( rowsToKeep < 0 || rowsToKeep > zeroone.length)
        throw new IndexOutOfBoundsException("row index " + rows) ;
    rows = rowsToKeep ;
    if ( zeroone.length > 0 )
        cols = zeroone[0].length ;
    else
        cols = 0 ;
    bits = new byte[rows][cols] ;
    n=0 ;
    /* clone the elements of zeroone (which may be modified later
```

```
    * by the calling program)
    */
    for (int r=0 ; r < rows ; r++)
        for (int c=0 ; c < cols ; c++)
        {
            bits[r][c] = (byte) zeroone[r][c] ;
            n += bits[r][c] ;
        }

    if (freep)
        reduce(Read) ;
} /* ctor */



/** Compute the perimeter.
* The perimeter is the sum of all edges of cells which are
* not shared with other cells. This includes the edges in cavities and holes.
* @return the perimeter (which is >=4 if there are any occupied cels)
* @since 2021-03-10
*/
public int perimeter()
{
    if ( n == 0 )
        return 0 ;
    /* the strategy is to start with the estimate 4*n
    * as if all celles were isolated, and to subtract 2
    * for all edges with 2 cells to the left and right or
    * above or below.
    */

    int per = 4*n ;
    /* loop over vertical internal edges.
    */
    for (int r=0 ; r < rows ; r++)
        for (int c=0 ; c < cols-1 ; c++)
        {
            if ( bits[r][c] >0 && bits[r][c+1] > 0)
                per -= 2 ;
        }

    /* loop over horizontal internal edges.
    */
    for (int c=0 ; c < cols ; c++)
        for (int r=0 ; r < rows-1 ; r++)
        {
            if ( bits[r][c] >0 && bits[r+1][c] > 0)
                per -= 2 ;
        }
    return per ;
} /* per */

/** Check whether the bit set has all 1 connected
* This is a static variant because creating a object of FreePoly-type
* would (usually) consider reduce() which is an expensive operation and not needed here.
* @param zeroone The binary array.
* @param validRows the array of zeroone has zeroone.length rows, but
*    only the rows 0..validRows are valid/known.
* @return True if the zeroone[0..validRows] first elements are connected.
* @since 2020-06-16
*/
public static boolean isConnected(final int [][] bits, int validRows)
{

    /* Rearrange the information of the occupied cells by putting
    * their 2D coordinates into a vector.
    */
    Vector<byte[]> freeSet = new Vector<byte[]>() ;
    for(int r=0 ; r < validRows ; r++)
        for(int c=0 ; c < bits[0].length ; c++)
            if ( bits[r][c] > 0 )
            {
```

```
            byte[] coo= new byte[2] ;
            coo[0] = (byte) r ;
            coo[1] = (byte) c ;
            freeSet.add(coo) ;
        }

/* the number of 1's in the zeroone[0..validRows] rows
*/
final int n = freeSet.size() ;

/* this set contains [r][c] lists of 2d cordinates
* of set bits (squares of the n-ominoe) connected
* with the first square. If all connections are checked,
* the size of this vector must be n if the n-ominoe is connected.
* We tackle the problem that some meandering forms of disconnected
* cell clusters do not define a n-ominoe. The algorithm is to
* put initially one of the freeSet[] cells into the cluster,
* and moving recursively the remaining freeSet[] cells into the
* cluster once it is verified that they share an edge with any of
* the cells in the cluster.
*/
Vector<byte[]> coneCluster = new Vector<byte[]>() ;
/* assume n>=1, so at least one element in freeSet()
*/
coneCluster.add(freeSet.firstElement()) ;
freeSet.removeElementAt(0) ;

for(; ! freeSet.isEmpty() ;)
{
    /* search through all freeSet squares and
    * try to add some to the connected cluster. Keep track
    * with the enlarged variable whether that succeeded for at
    * least one in the freeSet.
    */
    boolean enlarged = false ;
    for( byte[] cand: freeSet)
    {
        /* is this candidate neighbour of any in the cluster?
        */
        boolean isne = false ;
        for( byte[] inclus : coneCluster)
        {
            /* neighbour to the N, S, E or W: coordinate differences in the square lattice
            */
            if ( Math.abs(cand[0]-inclus[0]) == 1 && cand[1]==inclus[1]
                ||  Math.abs(cand[1]-inclus[1]) == 1 && cand[0]==inclus[0])
            {
                isne =true ;
                break ;
            }
        }
        if ( isne)
        {
            /* has  neibbour in the cluster: move from the
            * freeSet to coneCluster. It would be sufficient to use add()
            * here, but because the comparison in the isne loop starts at
            * the beginning of the coneCluster, and the initial freeSet was scanned
            * row-wise, it's faster to add these at the start...
            coneCluster.add(0,cand) ;
            */
            coneCluster.add(cand) ;
            freeSet.remove(cand) ;
            enlarged = true ;
            break ; /* needed to avoid scanning the moved elements */
        }
    }

    if ( ! enlarged)
        /* did not move any of the freeSet elements into the coneCluster
        */
        break ;
}
```

```
    /* connected, if all coordinate pairs have been moved from the freeSet
    * to the conCluster in the loop..
    */
    return ( coneCluster.size() == n ) ;
} /* isConnectedOld */


/**
* @brief construct the r-rooted free polyominoes
* @param rMult The number of marked cells. currently always r=1, the standard definition of rooted polyominoes.
* @return The free polyomioes where rooted cells are marked with a 2 in the matrix cell.
* @since 2019-05-25
*/
public Vector<FreePoly> rooted(int rMult)
{
    /* loop over all 1bits in the matrix, flip this to 2, reduce again, and
    * collect the distinct results.
    */
    Vector<FreePoly> rtd = new Vector<FreePoly>() ;
    for (int r=0 ; r < rows ; r++)
    for (int c=0 ; c < cols ; c++)
    {
        if ( bits[r][c] > 0 )
        {
            byte[][] bitsR = new byte[rows][cols] ;
            for (int rr=0 ; rr < rows ; rr++)
            for (int cc=0 ; cc < cols ; cc++)
                if ( rr == r && cc == c)
                    bitsR[rr][cc] = (byte)2 ;
                else
                    bitsR[rr][cc] = bits[rr][cc] ;

            /* construct a normalized rooted free polyomino
            */
            FreePoly mrkd = new FreePoly(bitsR,true,-1) ;
            boolean kown = false ;
            for( FreePoly k : rtd)
            {
                if ( compareTo(mrkd,k) == 0 )
                {
                    kown = true ;
                    break ;
                }
            }
            if ( ! kown )
                rtd.add(mrkd) ;
        }
    }
    return rtd ;
}

/** Construct the rotated and flipped versions. Retain only one.
* @param Read is a flag: if zero or positive, exclude 90degree rotation symmetries.
*  That means if negative, the equivalence operations of reducing
*  the set of occupied cells to a single representative are all tested,
*  of which there are 4 operations (group of identity, two flips and 180 deg rotation)
*  for rectangular shapes, and 8 operations (those by adding 90 degree rotations)
*  for the square shapes. If the Read parameter is zero or positive, that additional
*  set of 4 operations (90 degree rotations) is *not* added to the square shapes,
*  and all results will be blind to those symmetries, i.e., this is not what
*  the usual meaning for free polyominoes would do.
*/
private void reduce(int Read)
{
    /* if rows <> cols, compare this byte arry with the
    * three varians of flipped x, flipped y and rotated by 180 (group of order4).
    * If rows = cols, compare with the full D_8 group of order 8 by
    * including rotations by 90 or 270 degrees. piv is the pivotal variant
    * which is "smallest" in all the rotated/flipped variants.
    */
    byte[][] piv = bits ;
    byte[][] r180 = rot180(bits) ;
```

```
        byte[][] fpiv = flipx(bits) ;
        byte[][] fpiv180 = flipx(r180) ;
        if ( compareTo(r180,piv) < 0 )
            piv = r180 ;
        if ( compareTo(fpiv,piv) < 0 )
            piv = fpiv ;
        if ( compareTo(fpiv180,piv) < 0 )
            piv = fpiv180 ;
        if ( rows == cols && Read < 0)
        {
            /* consider 4 more versions if the matrix is square and
            * Read's table is not to be reproduced
            */
            byte[][] r90 = rot90(bits) ;
            if ( compareTo(r90,piv) < 0 )
                piv = r90 ;

            byte[][] r270 = rot90(r180) ;
            if ( compareTo(r270,piv) < 0 )
                piv = r270 ;

            byte[][] fpiv90 = flipx(r90) ;
            if ( compareTo(fpiv90,piv) < 0 )
                piv = fpiv90 ;

            byte[][] fpiv270 = flipx(r270) ;
            if ( compareTo(fpiv270,piv) < 0 )
                piv = fpiv270 ;
        }
        /* replace the representation by the "smallest" one.
        * Sum of parts, row and col are not changed by this representation.
        */
        bits = piv ;
    } /* reduce */


    /** Check whether the polynomino has a symmetry of order 2 etc.
    * @param symm String N, R, H, V or empty.
    *  The empty string means that we don't care aboutsymmetry and the result is always 'true.'
    * @return true if it has a N, R, H, V symmetry.
    */
    public boolean isSymm(String symm)
    {
        if ( symm.isEmpty() )
            return true ;

        if ( symm.indexOf("R") >=0 )
        {
            byte[][] r180 = rot180(bits) ;
            if ( compareTo(bits, r180) != 0 )
                return false ;
        }

        if ( symm.indexOf("H") >=0 )
        {
            byte[][] fl = flipy(bits) ;
            if ( compareTo(bits, fl) != 0 )
                return false ;
        }

        if ( symm.indexOf("V") >=0 )
        {
            byte[][] fl = flipx(bits) ;
            if ( compareTo(bits, fl) != 0 )
                return false ;
        }

        if ( symm.indexOf("N") >=0 )
        {
            /* request that this must not have the R, H or V symmetry...
            */
            byte[][] img = rot180(bits) ;
```

```
        if ( compareTo(bits, img) == 0 )
            return false ;
        img = flipx(bits) ;
        if ( compareTo(bits, img) == 0 )
            return false ;
        img = flipy(bits) ;
        if ( compareTo(bits, img) == 0 )
            return false ;
    }

    return true ;
}

/** Test whether another object is an polyominoe of the same arrangement of cells.
* @param oth The object to be compared to this.
* @return True if the object is a polyomino with the same shape and arrangement of cells.
*/
@Override
public boolean equals(Object oth)
{
    if ( oth instanceof FreePoly )
    {
        byte[][] othbits = ((FreePoly)(oth)).bits ;
        return ( compareTo(bits,othbits) ==0 ) ;
    }
    else
        return false;
}

/** Define a lexicographic order of 2D byte arrays by comparing them row by row
* @param left The first array to be considered.
*   Must be rectangular (must have the same number of elements in each row).
* @param right The second array to be considered.
*   Must be rectangular (must have the same number of elements in each row).
* @return a value of -1, 0 or +1 if left is considere to  be smaller than, equal to or larger than right.
*/
private static int compareTo( final byte[][] left, final byte[][] right)
{
    if ( left.length > right.length)
        return 1 ;
    else if ( left.length < right.length)
        return -1 ;
    else if ( left.length == 0 )
        return 0 ;
    else
    {
        if ( left[0].length > right[0].length)
            return 1;
        else if ( left[0].length < right[0].length)
            return -1;
        else if ( left[0].length == 0)
            return 0;
        else
        {
            final int rows =left.length ;
            final int cols =left[0].length ;
            for(int r=0 ; r < rows ; r++)
            for(int c=0 ; c < cols ; c++)
            {
                if ( left[r][c] > right[r][c])
                    return 1 ;
                else if ( left[r][c] < right[r][c])
                    return -1 ;
            }
            return 0 ;
        }
    }
}

/** Define a lexicograph order of fixed n-ominoes by comparing their binary matrix representations
* @param left The first polyomino.
* @param right The second polyomino.
```

```
* @return a value of -1, 0 or +1 if left is regarded to  be smaller, equal to or larger than right.
*/
static int compareTo( final FreePoly left, final FreePoly right)
{
    return compareTo(left.bits, right.bits) ;
}

/** Flip elements of array by swapping columns
* @return The clone of the byte array where within each row the order of elements is reversed
*/
static byte[][] flipx(final byte[][] in)
{
    final int rows = in.length ;
    final int cols = ( rows > 0 ) ? in[0].length : 0 ;
    byte[][] out = new byte[rows][cols] ;
    for(int r = 0 ; r < rows ; r++)
        for(int c=0 ; c < cols ; c++)
            out[r][c] = in[r][cols-1-c] ;
    return out ;
}

/** Flip elements of array by swapping rows
* @return The byte array where row r is swapped with row in.length-1-r.
*/
static byte[][] flipy(final byte[][] in)
{
    final int rows = in.length ;
    final int cols = ( rows > 0 ) ? in[0].length : 0 ;
    byte[][] out = new byte[rows][cols] ;
    for(int r = 0 ; r < rows ; r++)
        for(int c=0 ; c < cols ; c++)
            out[r][c] = in[rows-1-r][c] ;
    return out ;
}

/** Rotate array by 180 degrees
* @return The byte array flipped by y and then by x.
*/
static byte[][] rot180(final byte[][] in)
{
    final int rows = in.length ;
    final int cols = ( rows > 0 ) ? in[0].length : 0 ;
    byte[][] out = new byte[rows][cols] ;
    for(int r = 0 ; r < rows ; r++)
        for(int c=0 ; c < cols ; c++)
            out[r][c] = in[rows-1-r][cols-1-c] ;
    return out ;
}

/** Rotate array by 90 degrees ccw.
* @return The clone of the byte array where the number of columns and rows have been swapped.
*/
static byte[][] rot90(final byte[][] in)
{
    final int cols = in.length ;
    final int rows = ( cols > 0 ) ? in[0].length : 0 ;
    byte[][] out = new byte[rows][cols] ;
    for(int r = 0 ; r < rows ; r++)
        for(int c=0 ; c < cols ; c++)
            out[r][c] = in[c][rows-1-r] ;
    return out ;
}

/** @brief List occupied cells.
* This prints the array in a parenthetical list of the form (c1r,c1c)(c2r,c2c)(c3r,c3c),..
* where (cir,cic) are the row and column indices of the non-zero entries in the array.
* Both cir and cic and 0-based, i.e., start at 0 at the first row and first column.
* The order of the occpuied cells is row by row.
* @param in[][] A rectangular array of numbers
* @return A string representation of the coordinates of occupied cells.
* @since 2020-06-25
*/
```

```
    public static String toStringCells(final byte in[][])
    {
        final int rows = in.length ;
        final int cols = ( rows > 0 ) ? in[0].length : 0 ;
        String str = new String();
        /* first a single line of cell coordinates
        */
        for(int r=0 ; r < rows ; r++)
        {
            for(int c=0 ; c < cols ; c++)
                if ( in[r][c] != 0 )
                    str += "(" + r + "," + c + ")" ;
        }
        return str ;
    } /* toStringCells */

    /** Derive an ASCII representation of the polyomino.
    * @param in[][] A rectangular array of 1-digit numbers
    * @return A string representation of the occupied cells and a string representation of the binary matrix.
    * @since 2020-06-25 Add another  line with a parenthetical list of the occupied cells
    */
    public static String toString(final byte in[][])
    {
        final int rows = in.length ;
        final int cols = ( rows > 0 ) ? in[0].length : 0 ;
        /* start with a line of coordinate pairs of occupied cells
        */
        String str = toStringCells(in) ;
        str += System.getProperty("line.separator") ;

        /* Then the version of a binary matrix where a human can recognize the connectivity.
        * First row 0, then row 1 (to relate this to the info in the previous line)
        */
        for(int r=0 ; r < rows ; r++)
        {
            for(int c=0 ; c < cols ; c++)
                str += in[r][c] ;
            str += System.getProperty("line.separator") ;
        }
        return str ;
    } /* toString */



    /** Print a human-readable pattern of 0's and 1's that represent the polyomino.
    * @return The zeros and ones with one list per output line.
    */
    public String toString()
    {
        return toString(bits) ;
    } /* toString */


} /* FreePoly */
```

Appendix D. Source code of de/mpg/mpia/rjm/FreePolySet.java

```
/*
* $Header: de/mpg/mpia/rjm/FreePolySet.java$
*/

/** @file
* The set of n-ominoes with a fixed rows X cols bounding box.
* @author Richard J. Mathar
* @see <a href="http://vixra.org/abs/1905.0474">vixra:1905.0474</a>
*/

package de.mpg.mpia.rjm ;

import java.util.* ;
import java.lang.* ;
```

```
/**
* @brief compute the set of all free n-ominoes with given bounding rectangle.
* @since 2019-05-11
* @author Richard J. Mathar
*/
public class FreePolySet
{
    /** the sum of the parts
    */
    int n ;

    /** the number of rows
    */
    int rows ;

    /** the number of columns
    */
    int cols ;

    Vector<FreePoly> polys ;

    /** Flag to introduce intermedia percolation checks
    */
    boolean TEAROFF = false ;

    /** Flag to impose only a set of 4 (not 8) symmetry operations on free polyonomoes of square shape.
    * Use or not use C. Read's incomplete symmetries for polyominoes with square bounding rectangle.
    *    If this parameter is zero or positive, the reduction of free polyominoes
    *    to a single representative does not try reduction by 90 degree rotations of square shapes
    *    as in C. Read, doi:10.4153/CJM-1962-001-2, A059682, A059683, A059684.
    */
    static int CREAD = -1 ;

    /**
    * Constructor with a predefined n-omino.
    * This just stores the main parameters and does not actually
    * compute anything.
    * @param n The number of squares in each n-onmino
    * @param r The number of rows in each n-omino
    * @param c The number of columns in each n-omino
    * @since 2019-05-11
    */
    public FreePolySet(int n, int r, int c)
    {
        this.n = n ;
        rows = r ;
        cols = c ;
        polys = new Vector<FreePoly>() ;
    } /* ctor */

    /** Main part of the solution: create all n-ominoes.
    * @param freep If true create free polyominoes, else fixed.
    * @param s A string of N, H, ,V, R symmetry requirements. Empty if none.
    * @param jThrd Number of parallel threads generating the free n-ominoes
    */
    public void create(boolean freep, final String symm, int jThrd) throws InterruptedException
    {
        /* no solution if there are more cells n than r*c cells in the rectangle.
        */
        if ( n > rows*cols)
            return ;

        /* Each row must contain at least one square to
        * keep the n-omino connected, and at most cols squares because
        * the columns are essentially bitsets. Create all compositions of n
        * into 'rows' parts, each part in the range 1..cols.
        */
        Composit rowComp = new Composit(n,rows,1,cols) ;

        if ( rowComp.size() <= 0 )
            return ;
```

```
    /* Compute only once all possible bit sets of the rows.
    * bitsets[i] contains the bitsets with i bits set, and each bitset with 'cols' bits.
    * To simplify the indexing, the sum is also performed for bweit=0, although
    * this gives actually only the trivial all-0 list which cannot occur if n>=1.
    */
    Vector<Composit> bitsets = new Vector<Composit>() ;
    for (int bweit =0 ; bweit <= cols ; bweit++)
    {
        Composit hamm = new Composit(bweit,cols,0,1) ;
        bitsets.add(hamm) ;
    }

    /* prepare threads to run. We will execute one or more row compositions
    * in a single thread, so it would be useless to generate more threads than
    * the current number of compositions...
    */
    jThrd = Math.min(jThrd,rowComp.size()) ;

    FreePolySetThrd[] thrds = new FreePolySetThrd[jThrd] ;
    Thread[] thrdsT = new Thread[jThrd] ;
    for (int m=0 ; m < jThrd ; m++)
    {
        thrds[m] = new FreePolySetThrd(n, rows, cols, freep, CREAD, symm, rowComp, bitsets, m, jThrd) ;
        thrdsT[m] = new Thread(thrds[m]) ;
        thrdsT[m].start() ;
    }

    /* wait for all threads to finish */
    for(int m=0 ; m < jThrd; m++)
    {
        thrdsT[m].join(0) ;
    }

    /* collect all the polyominioes of the threads into a single vector.
    */
    for(int m=0 ; m < jThrd; m++)
    {
        if ( rows != cols)
            polys.addAll( thrds[m].polys) ;
        else
        {
            /* Those of square shape may have
            * been generated by a row sum and also by another row sums (treated as a column sum).
            * These are to be reduced to generate them only once.
            */
            for( FreePoly p: thrds[m].polys)
            {
                if ( ! polys.contains(p) )
                    polys.add(p) ;
            }
        }
    }

} /* create */


/** List all polyominoes in the set
* @return The binary vectors [c00,c01...],[c10,c11...]
*/
public String toString()
{
    return toString(polys) ;
} /* toString */

/** List all polyominoes in the set
* @return The binary vectors [c00,c01...],[c10,c11...]
*/
public static String toString(Vector<FreePoly> polys)
{
    String str = new String() ;
    for (int i=0 ; i < polys.size() ; i++)
        str += polys.elementAt(i).toString() + "\n" ;
```

```
        return str ;
    } /* toString */

    /** Count the number of polyominos in the set with given perimeter
    * @param per The perimeter (an even integer).
    * @return the number of elements in polys with exactly perimeter per.
    * @since 2021-03-10
    */
    public int cntPerim(int per)
    {
        if ( per < 4 || (per %2) != 0 || per > 2*n+2)
            return 0 ;

        int ct= 0 ;
        for( FreePoly p: polys)
            if ( p.perimeter() == per )
                ct ++ ;
        return ct ;
    } /* cntPerim */

    /** Main program
    * usage: java -cp . FreePolySet [-v] [-f] [-h #height] [-r R] [-w #width] [-s symm] [-j Nthreadds] #size
    */
    public static void main(String[] args) throws InterruptedException
    {
        /* if verb=true, print also the {0,1} matrices
        */
        boolean verb = false ;

        /* if freep=true, generate only free polynomios, else fixed
        */
        boolean freep = true ;

        /* The number of rooted cells. Initially 0 (=not rooted) for A000105.
        * Using rMult=1 (with the option -r 1) gives OEIS A126202, the "pointed" polyominoes with n cells.
        */
        int rMult = 0 ;

        /* If this is a nonnegative integer: consider only polyominoes with that specific
        * number of columns (=width)
        */
        int fixedCol = -1 ;

        /* If this is a nonnegative integer: consider only polyominoes with that specific
        * number of rows (=height)
        */
        int fixedRow = -1 ;

        /* number of threads run in parallel
        */
        int jThrd =1 ;

        /* If not empty: count only polyominoes with a symmetry.
        * N=none, R=rotation by 180 degrees, H=rotation along the short axis, V=rotation along long axis.
        * See Redelmeister.
        * Note that these are minium requirements. There is for example the full-block polyominos
        * which has H+V+R.
        * There is no implementation here to check for symmetry along the 2 diagonals for
        * the polyominoes with a square hull.
        */
        String symm = new String() ;

        /* empty list of arguments: usage hint
        */
        if ( args.length == 0 )
        {
            final FreePolySet tmp = new FreePolySet(0,0,0) ;
            System.out.println("Usage: java -cp . " + tmp.getClass().getName()
                + " [-j Nthrd] [-w width] [-v] [-s Symm] ncell") ;
            return ;
        }
```

```
for( int optind =0 ; optind < args.length ; optind++)
{
    if ( args[optind].equals("-v") )
        verb =true ;
    if ( args[optind].equals("-f") )
        freep =false  ;
    if ( args[optind].equals("-r") )
        rMult = Integer.parseInt(args[++optind]) ;
    if ( args[optind].equals("-w") )
    {
        fixedCol = Integer.parseInt(args[++optind]) ;
        /* correct if erroneous non-positive input
        */
        fixedCol = Math.max(fixedCol,1) ;
    }
    if ( args[optind].equals("-h") )
    {
        fixedRow = Integer.parseInt(args[++optind]) ;
        /* correct if erroneous non-positive input
        */
        fixedRow = Math.max(fixedRow,1) ;
    }
    if ( args[optind].equals("-j") )
    {
        jThrd = Integer.parseInt(args[++optind]) ;
        /* correct if erroneous non-positive input
        */
        jThrd = Math.max(jThrd,1) ;
    }
    if ( args[optind].equals("-s") )
        symm = args[++optind] ;
}

/* last command line argument is the polyomino size
*/
int n = Integer.parseInt(args[args.length-1]) ;

/* counter for the number of polyominoes in that class
*/
int tot = 0 ;

/* loop over all numbers of columns (=widths)
*/
for(int c= 1; c<=n; c++)
{
    /* if a request for only a single width (column) is made,
    * skip all others.
    */
    if ( fixedCol >= 0 && c != fixedCol)
        continue ;

    /* Loop over all row lengths (=heights).
    * Need r*c >= n, so don't start at 1..
    */
    int rmin = (CREAD>0 || !freep) ? 1 : Math.max(n/c,c) ;
    for(int r= rmin ; r+c-1 <=n ; r++)
    {
        if ( fixedRow > 0 && r != fixedRow)
            continue ;
        FreePolySet po = new FreePolySet(n,r,c) ;
        po.create(freep,symm,jThrd) ;
        if ( rMult > 0 && freep)
        {
            /* generate the rMult-rooted free polyominoes
            */
            Vector<FreePoly> rtd = new Vector<FreePoly>() ;
            for ( FreePoly nonrtd : po.polys)
            {
                rtd.addAll( nonrtd.rooted(rMult) ) ;
            }

            if ( rtd.size() > 0 )
```

```
                    {
                        if ( verb)
                            System.out.println( FreePolySet.toString(rtd) ) ;

                        System.out.println(""+ n + " " + r + " " + c + " : " + rtd.size()) ;
                        tot += rtd.size() ;
                    }
                }
                else
                {
                    if ( verb)
                        System.out.println(po.toString()) ;

                    if ( po.polys.size() > 0 )
                    {
                        /* since 2021-03: print also a statistics refined
                        * by perimeter to generate OEIS A342243
                        */
                        for(int per =0 ; per <= 2*n+2 ; per += 2)
                        {
                            final int byp = po.cntPerim(per) ;
                            if ( byp > 0 )
                                System.out.println(""+ n + " " + r + " " + c + " " + per + " : " + byp );
                        }
                        System.out.println(""+ n + " " + r + " " + c + " : " + po.polys.size()) ;
                        tot += po.polys.size() ;
                    }
                }
            }
        }
        if ( fixedCol < 0)
            System.out.println(""+ n + " : " + tot) ;
    } /* main */
} /* FreePolySet */
```

Appendix E. Source code of de/mpg/mpia/rjm/FreePolySetThrd.java

```
/*
* $Header: de/mpg/mpia/rjm/FreePolySetThrd.java$
*/

/** @file
* Construct the set of n-ominoes with a r X c bounding box and fixed n with a fixed number of threads.
* @author Richard J. Mathar
* @see <a href="http://vixra.org/abs/1905.0474">vixra:1905.0474</a>
*/

package de.mpg.mpia.rjm ;

import java.util.* ;
import java.lang.* ;

/**
* @brief compute a subset of all free n-ominoes with given bounding rectangle.
* @since 2020-06-22
* @author Richard J. Mathar
*/
public class FreePolySetThrd implements Runnable
{
    /** The number of cells in the n-omino and also the partition of the row sums.
    */
    int n ;

    /** The number of rows in each n-omino.
    */
    int rows ;

    /** The number of columns in each n-omino.
    */
    int cols ;

    /** Number of parallel threads generating the free n-ominoes
```

```
*/
int jThrd ;

/** This thread index, from 0 to Thrd-1.
*/
int mThrd ;

/** The polyominoes created by this thread
*/
Vector<FreePoly> polys ;

/** Flag to indicate if free (true) or fixed (false) n-ominoes are created
*/
boolean freep ;

/** Use C. Read incomplete symmetries for polyominoes with square bounding rectangle.
*/
int Read ;

/** A string with letters N, H, V, R symmetry requirements.
* If the string is empty, no such filtering by symmetry is done (but of course
* the free polyominoes are reduced to a single representative always with
* symmetry operations of flips taken into account).
*/
String symm ;

/** The compositions of the row sums of this thread. Each polyomino
* is a rows x cols matrix of 0's and 1's. rc[i] is the predefined row sum of row i.
* Thread number m works on the row sums m, m+jThrd, m+2*jThrd etc. Because
* these are disttinc if the free polyominoes are distinct, the sets of free
* polyominoes of the different threads are non-overlapping.
*/
Composit rowComp ;

/** the row sum number i is flushed out into a bitset to generate
* values of 0's and 1's for row number i. All these bitsets are
* initially computed only once in advance to save some time, supposing
* that row sums are not unique in the matrices of 0's and 1's.
*/
Vector<Composit> bitsets ;

/** Flag to introduce intermediate percolation checks.
* Heuristically it does not seem to speed up the computation, at least for n up to 13,
* so it is disabled here by default.
*/
boolean TEAROFF = false ;

/**
* Constructor that defines the task to be run.
* This just stores the main parameters and does not actually
* compute anything.
* @param n The number of squares/cells in each n-onmino
* @param r The number of rows in each n-omino
* @param c The number of columns in each n-omino
* @param freep If this is true, generate/count free polyomioes, else fixed polyominoes.
* @param CReadflag Use C. Read incomplete symmetries for polyominoes with square bounding rectangle.
* @param symmFilt A string with letters N, H, V and/or R to filter symmetric polyominoes.
*   If N is present, count/create only polys without symmetry (symmetry group has only
*   the identity). If H and/or V demand symmetry with respect to horizontal or
*   vertical flips, if R demand symmetry with respect to rotation by 180 degrees.
*   (Obviously taking H and V implies that R is also in the group, but not vice versa...)
*   If empty, count polymonoes indepdenent of symmetry.
* @param rsums The compositions of n into r parts. Each composition
*     defines a layout for row sums of the binary matrix of a polyomino.
* @param bits01 The bits01(s) contains the compositions of s in c parts,
*     each part either 0 or 1 (so these are bitsets).
* @param m Thread number in the range 0..j-1.
* @param j Total number of treads to be called.
* @since 2020-06-22
*/
public FreePolySetThrd(int n, int r, int c, boolean freep, int CReadflag,
        final String symmFilt, Composit rsums, Vector<Composit> bits01, int m , int j)
```

```
{
    this.n = n ;
    rows = r ;
    cols = c ;
    this.freep = freep ;
    Read = CReadflag ;
    symm = symmFilt ;
    polys = new Vector<FreePoly>() ;
    rowComp = rsums ;
    bitsets =bits01 ;
    mThrd = m ;
    jThrd = j ;
} /* ctor */

/** Main part of the solution: create n-ominoes where the rowComp's index is assigned to this thread.
*/
public void run()
{
    /* no solution if there are more cells n than r*c cells in the rectangle.
    */
    if ( n > rows*cols)
        return ;

    if ( rowComp.size() <= 0 )
        return ;

    /* Outer loop over all compositions of the row sums .
    * Select the ones to be handled by this thread. By using a simple modulo
    * distribution to disperse the threads, this should be like well-balanced,
    * because the row compositions have some regular distribution...
    */
    for ( int j = mThrd ; j < rowComp.comps.size() ; j += jThrd)
    {
        final int[] rc = rowComp.comps.elementAt(j) ;
        /* skip those where the reverse of the composition would be larger,
        * because we'll create them anyway by the 180 deg rotations...
        * Note that using the completementary set where compareTo(rc,rcrev)>=0
        * gives the same results, but is slower, because the bit sets of
        * larger weight in the initial rows then, and there is better efficiency
        * of using the restricition of connectivity.
        */
        final int[] rcrev = Composit.reverse(rc) ;
        if ( Composit.compareTo(rcrev,rc) >= 0  || !freep)
        {
            /* now row sums are fixed ; inner loop: distribute them over
            * rows: need binary vectors with rc[] bits set.
            */
            int[][] bits = new int[rows][cols] ;
            create(bits,0,rc) ;
        }
    }
} /* run */

/** Main part of the calculation: create all of them
* @param bits The polyomino with a bit[r][c] equal to one of the square is covered.
* @param prow The pivotal row from 0 up to the number of rows (-1 in Java).
*         This is the row in bits[][] which needs to be filled next.
* @param rc The vector of row sums. rc[r] is the number of bits to be set in row r.
*/
public void create(int[][] bits, int prow, int [] rc)
{
    /* impossible to create solutions if that row sum is larger than
    * the number of columns.
    */
    if ( rc[prow] > cols)
        return ;

    /* strategy is to find the bitsets that have as many
    * bits set as rc[prow] indicates. Connectivity: Check each of them
    * in turn if that has at least one common edge with the previous
    * row of bits (ie. bit-wise and is not zero), and preliminarily
    * add this as a new row.
```

```
*/
final Composit thisrow = (bitsets == null) ?
    new Composit(rc[prow],cols,0,1) : bitsets.elementAt(rc[prow]) ;
for( int[] brow : thisrow.comps)
{
    /* is the connectivity (percolation requirement) satisfied ?
    * percol=true if it is as defined for polyominoes.
    */
    boolean percol ;
    /* no contraint on bitset if this is the first row.
    */
    if ( prow == 0 )
    {
        percol = true;
        /* if this is for free polyominoes, we only need to start
        *with approximately the smaller "half" of the bitsets because
        * the other polyominoies can be created by flipping along the horiz. axis.
        * Skip dealing with this set brow[] of bits if the reversed
        * would be lexicographically smaller.
        */
        if ( freep)
        {
            final int[] bitsRev = Composit.reverse(brow) ;
            if ( Composit.compareTo(bitsRev, brow) < 0 )
                continue ;
        }
    }
    else
    {
        percol = false;

        /* run with a bit (column) wise and along the columns and
        * check that at least one of the squares is edge-connected with
        * a square of the previous row
        */
        for(int c =0 ; c < cols && !percol; c++)
        {
            if ( brow[c] == 1 && bits[prow-1][c] == 1)
                percol = true ;
        }

    }

    /* continue only if connectivity with adjacent row is verified
    */
    if ( percol)
    {
        /* copy selected bitset into the current pivotal row
        */
        bits[prow] = brow ;

        if ( prow == rows-1)
            /* reached a leave of the search scan: all bits[][] now defined.
            */
            add(bits) ;
        else
        {
            if ( TEAROFF && ( prow > 0 ) && ( prow % 5 == 0) )
            {
                /* cover the array of bits with a top layer of all-1
                * as if optimistic that there will be some arching cluster of
                * 1's added latter to connect the pieces. Note that prow+1 cannot
                * exceed rows here because that's caught in the if-clause above.
                */
                int roweff ;
                if ( Arrays.equals(bits[prow], bitsets.elementAt(cols).comps.elementAt(0)) )
                {
                    roweff = prow+1 ;
                }
                else
                {
                    bits[prow+1] = bitsets.elementAt(cols).comps.elementAt(0) ;
```

```
                        roweff = prow+2 ;
                    }
                    if ( ! FreePoly.isConnected(bits, roweff) )
                    {
                        continue ;
                    }
                }
                /* recursively add next adjacent row */
                create(bits, prow+1,rc) ;
            }
        }
    }
} /* create */

/** Check whether bits[][] is a valid n-omino.
* Add to the list if not yet present.
* @param bits The 2D bit set with 1's for occupied and 0's for unoccpied celles.
*/
void add(int[][] bits)
{
    /* Check that all parts of the composition of the column sums are >0
    * (no shotgun solutions admitted..)
    */
    for(int c=0 ; c < cols ; c++)
    {
        int su = 0 ;
        for(int r=0 ; r < rows ;r++)
            su += bits[r][c] ;
        if ( su == 0 )
        {
            return ;
        }
    }

    if ( FreePoly.isConnected(bits,rows) )
    {
        /* create a candidate polyomino for insertion, normalized representation
        */
        FreePoly cand =new FreePoly(bits,freep,Read) ;
        /* check whether this fails to be in any symmetry class that might be requested
        */
        if ( ! cand.isSymm(symm) )
            return ;
        /* check wheter this is a new n-omino.
        * Append the new polyomino if it differs from all the known ones.
        */
        if ( ! polys.contains(cand) )
            polys.add(cand) ;
    }
} /* add */

} /* FreePolySetThrd */
```

References

1. Edward A. Bender, L. Bruce Richmond, and S. G. Williamson, *Central and local limit theorems applied to asymptotic enumeration. iii. matrix recursions*, J. Combin. Theory A **35** (1983), no. 3, 263–278. MR 0721368
2. O. E. I. S. Foundation Inc., *The On-Line Encyclopedia Of Integer Sequences*, (2021), https://oeis.org/. MR 3822822
3. Iwan Jensen, *Counting polyominoes: a parallel implementation for cluster computing*, Lect. Notes Comp. Science **2659** (2003), 203–212.
4. David A. Klarner, *Some results concerning polyominoes*, Fib. Quart. **3** (1965), no. 1, 9–20. MR 0186569
5. ———, *Cell growth problems*, Canad. J. Math. **19** (1967), 851–863. MR 0214489
6. Donald E. Knuth and Richard Stong, *Problem 10875, animals in a cage*, Am. Math. Monthly **110** (2003), no. 3, 243–245.

7. Sascha Kurz, *Convex hulls of polyominoes*, Beitr. Algebra Geom. **49** (2008), no. 1. MR 2410568
8. ———, *Counting polyominoes with minimum perimeter*, Ars Combinatoria **88** (2008), 161–174. MR 2426414
9. ———, *Counting polyominoes with minimum perimeter*, arXiv:math/0506428 (2015).
10. Wolfgang R. Müller, Klaus Szymanski, Jan V. Knop, and Nenad Trinajstic, *On the number of square-cell configurations*, Theor. Chim. Acta **86** (1993), 269–278.
11. T. R. Parkin, L. J. Lander, and D. R. Parkin, *Polyomino enumeration results*, SIAM review, vol. 10, SIAM Fall Meeting, no. 2, 1967, pp. 244–290.
12. Ronald C. Read, *Contributions to the cell growth problem*, Canad. J. Math. **14** (1962), no. 1, 1–20. MR 0131367
13. D. Hugh Redelmeier, *Counting polyominoes: Yet another attack*, Disc. Math. **36** (1981), no. 2, 191–203. MR 0675352
14. Arne Schmidt, Sheryl Manzoor, Li Huang, Aaron T. Becker, and Sándor P. Fekete, *Efficient parallel self-assembly under uniform control inputs*, IEEE Robot. Autom. Lett. **3** (2018), no. 4, 3521.

Email address: `mathar@mpia-hd.mpg.de`
URL: `https://www.mpia-hd.mpg.de/~mathar`

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