ILLUSTRATIONS OF n-OMINO "BEST" COVERS WITH STRAIGHT k-OMINOS (A308437)

RICHARD J. MATHAR

Abstract. A308437 $T = (n, k)$ shows in how many ways a free n-omino can be covered by as many straight k-ominoes as possible.

1. Definitions

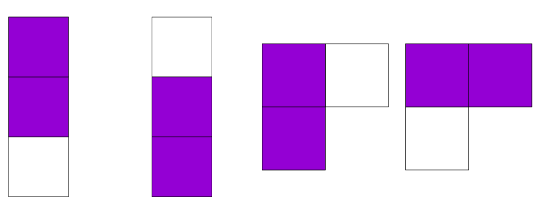
This is defined by the following covering problem:

(1) For each of the free n-ominoes consider one fixed representative.
(2) Determine how many nonoverlapping straight k-ominoes can be placed at maximum into this representative. Call this maximum $m(i, n, k)$ for the representative of the i-th n-omino. $m(i, n, k) \le n/k$. *Placing into* means that each cell of the m k-ominoes must be a cell of the representative.
(3) Determine all $T(i, n, k)$ configurations with the same $m(i, n, k)$, considering configurations distinct if any perimeter of one of the k-ominoes changes. That means if configurations were the same under the symmetry group of the n-omino, count them still as being distinct.
(4) Sum up all the configurations, $T(n, k) = \sum_{i=1}^{A105(n)} T(i, n, k)$.

The trivial counts are

- $T(n, 1)$ =A000105(n), the tiling of any n-omino by single squares,
- and $T(n, n) = 1$, the tiling the single straight n-omino by itself.

2. $n = 3$

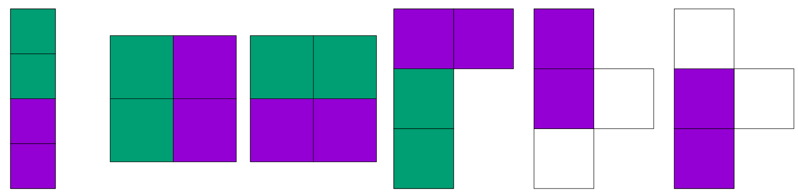


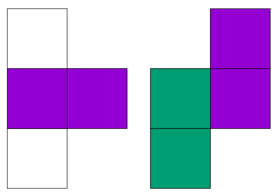
2.1. $T(3, 2) = 4$.

Date: May 27, 2019.
2010 *Mathematics Subject Classification.* Primary 05B50; Secondary 05A18, 51-04.
Key words and phrases. Free Polyomino, Incomplete Tiling.

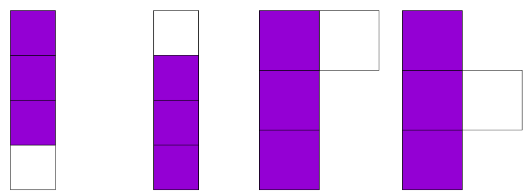
3. $n = 4$

3.1. $T(4, 2) = 8$.



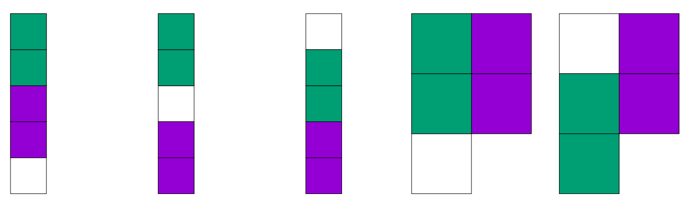


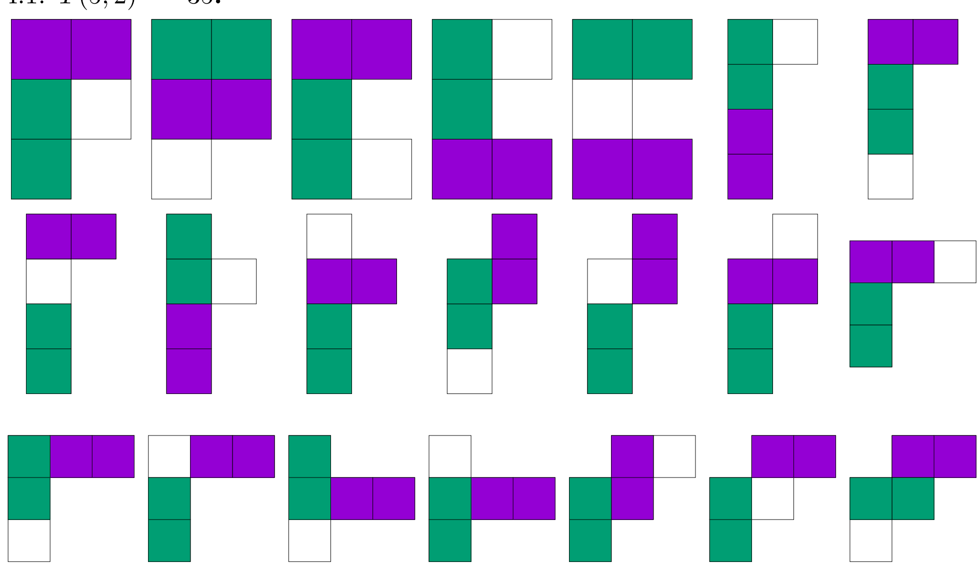
3.2. $T(4, 3) = 4$.

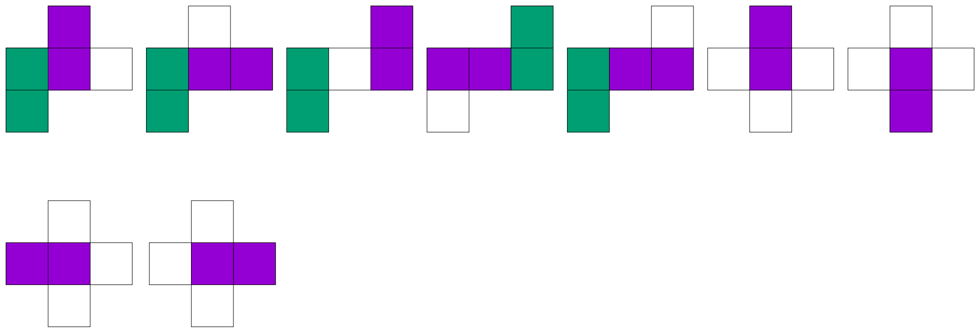


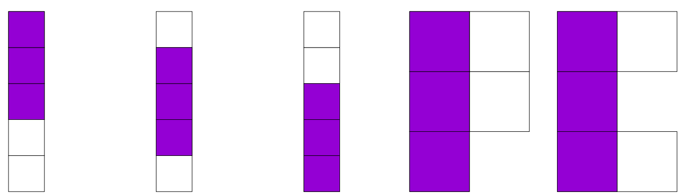
4. $n = 5$

4.1. $T(5, 2) = 35$.

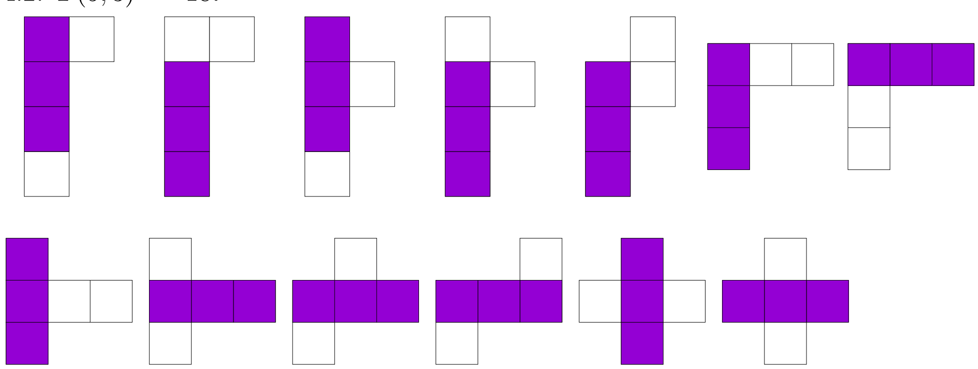


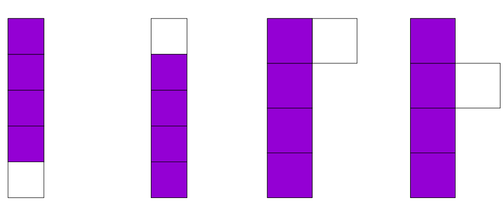






4.2. $T(5,3) = 18$.

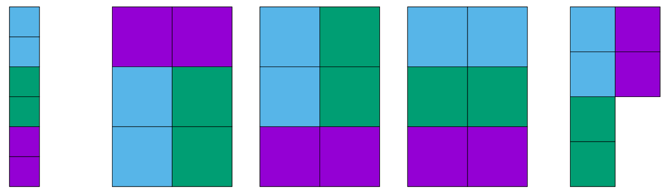




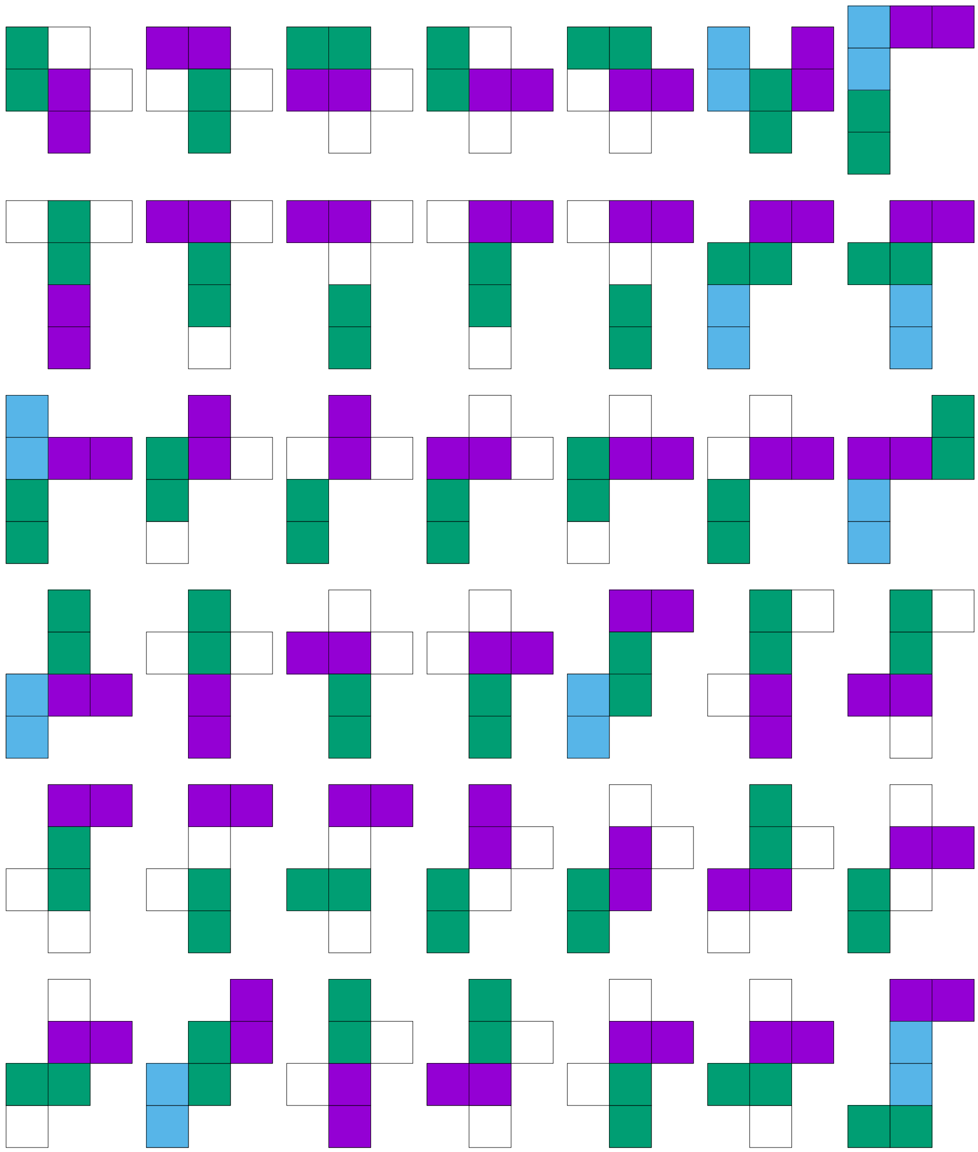
4.3. $T(5,4) = 4$.

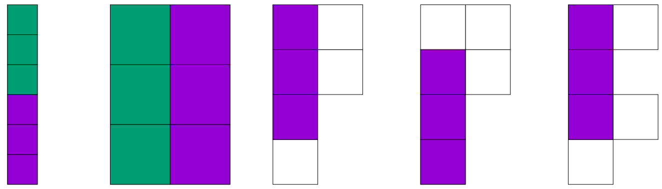
---

5. $n = 6$

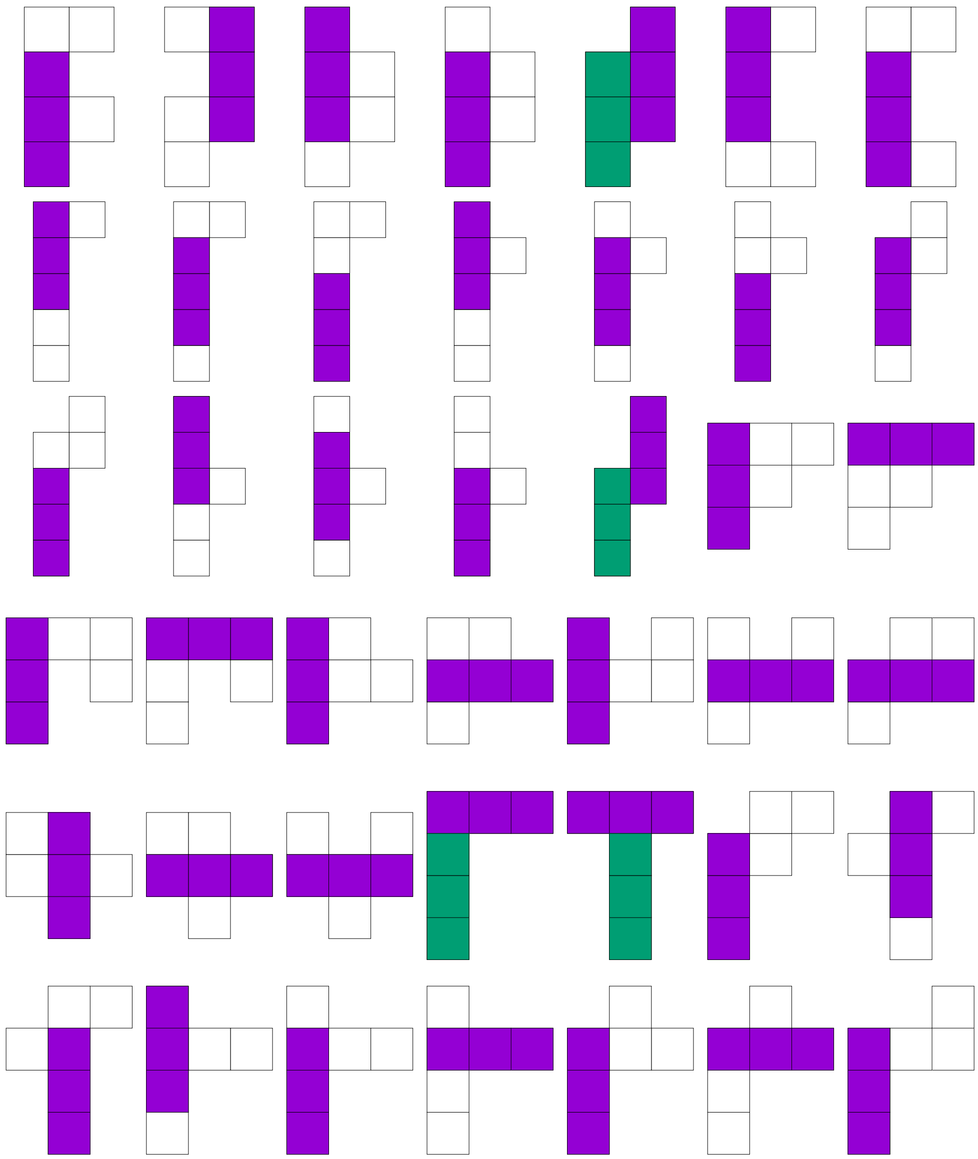


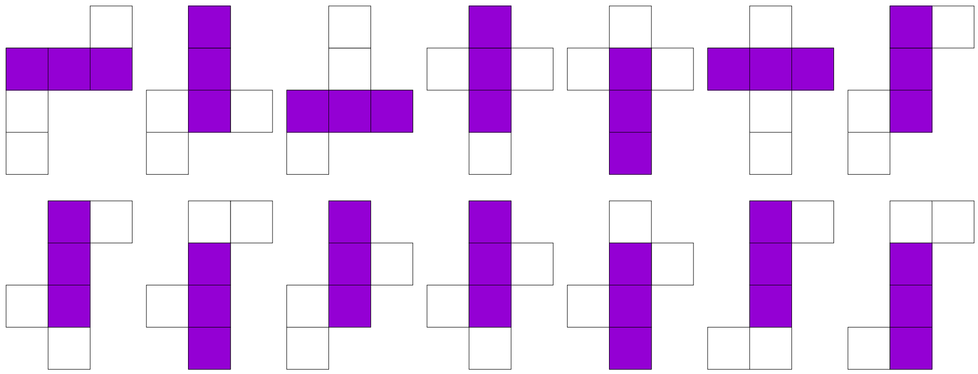
5.1. $T(6,2) = 89$.



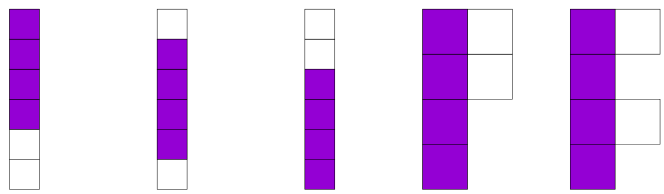


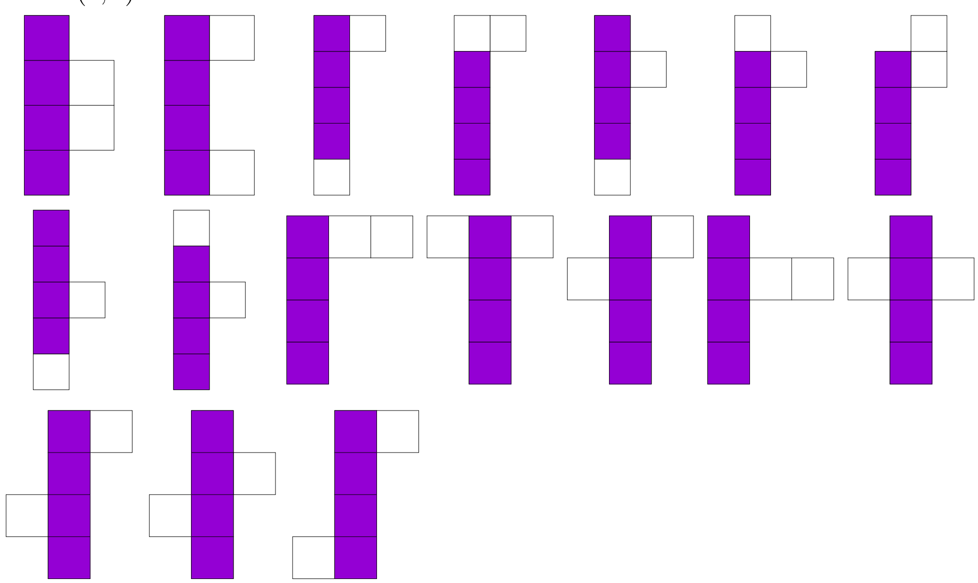
5.2. $T(6,3) = 61$.



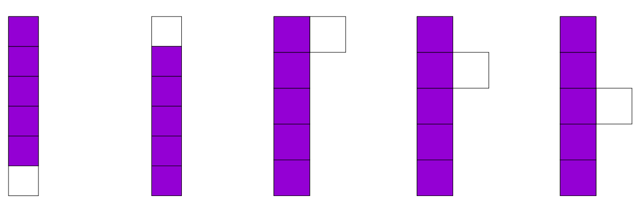


5.3. $T(6,4) = 22$.





5.4. $T(6,5) = 5$.



Appendix A. Brute Force Algorithm

The Java source program is reproduced here. This is derived from the free n-omino generation program represented in vixra:1905.0474. In a first step the set of free n-ominoes is generated. Then k is fixed, and for the horizontal and vertical variants of the straight k-omino we generate the set of k-ominoes that can be placed inside the n-omino. (In a simple loop move the lower or left pivotal cell of the k-omino through all the cells of the n-omino and discard the placements of the k-omino where some of the cells of the k-omino are not inside the n-omio.)

In loop over the multisets of these "compatible" fixed k-ominoes, starting with the sets of large order and counting down, figure out whether these k-ominoes have any pair-wise overlap. If they do not overlap, exit the loop over the multisets because $m(i, n, k)$ is found, the number of k-ominoes in that multiset.

Start from $T(n, k) = 0$ and for each multiset of order $m(i, n, k) \geq 1$ of nonoverlapping k-ominoes add one.

The code is compiled with

```
javac -cp . Composit.java FreePoly.java FreePolySet.java
```

and then called with two parameters, the first $n \geq 1$ and the second $1 \leq k \leq n$, as

```
java -cp . FreePolySet n k
```

Appendix B. Source code of Composit.java

```
/** @file
* A class which generates the compositions of an integer into a fixed number of parts.
* @author R. J. Mathar
*/

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

/**
* @brief The set of compositions of some fixed positive integer.
* @since 2019-05-11
*/
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
    * @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 differnt 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 */


} /* class Composit */
```

Appendix C. Source code of FreePoly.java

```
/** @file
* A n-omino with a r times c bounding box.
* @author R. J. Mathar
*/

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

/**
* @brief A free n-omino with a tight bound box of r rows and c columns.
* @since 2019-05-11
* @author R. 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.
    * @since 2019-05-11
    */
    public FreePoly(final byte[][] zeroone)
    {
        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] ;
        }

        reduce() ;
    } /* ctor */

    /**
    * Constructor with a fixed straight k-omino
    * @param x 0-based horizontal origin of the k-omino
    * @param y 0-based vertical origin of the k-omino
    * @param k length of the k-omino
    * @param horiz If true horizontal, else vertical
    */
    public FreePoly(int x, int y, int k , boolean horiz)
    {
        n=k ;
        if ( horiz)
        {
            cols = x+k ;
```

```
            rows = y+1 ;
            bits = new byte[rows][cols] ;
            for (int c=0 ; c < k ; c++)
                bits[y][x+c] = (byte)1 ;
        }
        else
        {
            cols = x+1 ;
            rows = y+k ;
            bits = new byte[rows][cols] ;
            for (int r=0 ; r < k ; r++)
                bits[y+r][x] = (byte)1 ;
        }

    } /* ctor */

    /** check whether the occupied cells are al inside the host's cells set
    * @param host The polyomino that is supposed to have a superset of this ominoes cells.
    */
    public boolean inside(FreePoly host)
    {
        for (int r=0 ; r < rows ; r++)
        for (int c=0 ; c < cols ; c++)
        {
            if ( bits[r][c] > 0 )
            {
                if ( r >= host.rows || c >= host.cols)
                    return false;
                if ( host.bits[r][c] == 0 )
                    return false ;
            }
        }
        return true ;
    }

    /** detect wheter a pair of the polyomioes overlaps
    * @param oth The polyomino that many overlap partially with this.
    * @return True if any pair of cells occurs in this and also in oth.
    */
    boolean overlaps(FreePoly oth)
    {
        for (int r=0 ; r < rows ; r++)
        for (int c=0 ; c < cols ; c++)
        {
            if ( bits[r][c] > 0 )
            {
                if ( r < oth.rows && c < oth.cols)
                {
                    if ( oth.bits[r][c] > 0 )
                        return true ;
                }
            }
        }
        return false ;
    }

    /** Construct the rotated and flipped versions. Retain only one.
    */
    private void reduce()
    {
        /* 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[][] r90 = rot90(bits) ;
        byte[][] r180 = rot90(r90) ;
        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 )
    {
        /* consider 4 more versions if the matrix is square
        */
        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 */

/** 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 ;
}

/** 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 ;
}

/** List all compositions
* @param in[][] A rectangular array of 1-digit numbers
* @return A string representation of the vectors [c00,c01...],[c10,c11...] separated by line feeds.
*/
public static String toString(final byte in[][])
{
    final int rows = in.length ;
    final int cols = ( rows > 0 ) ? in[0].length : 0 ;
    String str = new String();
    for(int r=0 ; r < rows ; r++)
    {
        for(int c=0 ; c < cols ; c++)
            str += in[r][c] ;
        str += "\n" ; /* todo: use locale line feed */
    }
    return str ;
} /* toString */

/** Test whether any pair of the polyomioes in the pset overlap
* @pset A set of polyominoes.
* @return True if at least one pair in pset has a common cell.
*/
static boolean overlap(Vector<FreePoly> pset)
{
    /* loop over all pairs
    */
    for(int i=0 ; i < pset.size() -1 ; i++)
        for(int j=i+1; j < pset.size() ; j++)
        {
            if ( pset.elementAt(i).overlaps(pset.elementAt(j)) )
                return true;
        }
    return false ;
}

/** Fill it with a maximum of straight k-ominoes.
```

```
* @param k The lenght of the straight filler-omino.
* @return The set of all distinct fixed straight k-ominoes that can be placed inside this polyomino.
*/
private Vector<FreePoly> inlets(int k)
{
    Vector<FreePoly> kpolys = new Vector<FreePoly>() ;
    for(int x=0 ; x < cols ; x++)
    for(int y=0 ; y < rows ; y++)
    {
        FreePoly cand = new FreePoly(x,y,k,true) ;
        if ( cand.inside(this))
            kpolys.add(cand) ;
    }
    if ( k > 1 )
    {
        for(int x=0 ; x < cols ; x++)
        for(int y=0 ; y < rows ; y++)
        {
            FreePoly cand = new FreePoly(x,y,k,false) ;
            if ( cand.inside(this))
                kpolys.add(cand) ;
        }
    }
    return kpolys ;
}

/** Select a maximum subset of non-intersectino inlets.
* @param inlets the set of individual k-monios that can be placed inside this.
* @return The maximum number of elements of inlets that do not overlap.
*/
private int maxinlets(final Vector<FreePoly> inlets)
{
    /* run over the bisets that select a subset of inlets
    */
    for(int sel= inlets.size() ; sel >= 0 ; sel--)
    {
        Composit selbits = new Composit(sel, inlets.size(), 0, 1) ;
        for( int[] bsel : selbits.comps)
        {
            /* generate a subset of the inlets elements */
            Vector<FreePoly> inlsub  = new Vector<FreePoly>();
            for(int s=0 ; s < bsel.length ; s++)
                if (bsel[s] > 0 )
                    inlsub.add(inlets.elementAt(s) );

            /* if not overlapping: exit early toindicate max subset size
            */
            if ( ! overlap(inlsub) )
                return sel ;
        }
    }
    return 0 ;
}

/** Create all ways of filling straight k-minos of maximum area in the sense of A308437.
* @param k The length of the straight k-ominoes that are to be placed.
* @param verb If true construct a format which is used for plotting.
* @return The number of configurations with a maximum degree of filling with nonoverlapping straight k-ominoes.
*/
int fill(int k, boolean verb)
{
    int configs =0 ;
    Vector<FreePoly> kstr = inlets(k) ;
    int kmax = maxinlets(kstr) ;
    /* rerun maxinlets() for that size but now gereate all configs
    */
    Composit selbits = new Composit(kmax, kstr.size(), 0, 1) ;
    for( int[] bsel : selbits.comps)
    {
        /* generate a subset of the inlets elements */
        Vector<FreePoly> inlsub  = new Vector<FreePoly>();
        for(int s=0 ; s < bsel.length ; s++)
```

```
                if (bsel[s] > 0 )
                    inlsub.add(kstr.elementAt(s) );

            /* if not overlapping: exit early toindicate max subset size
            */
            if ( inlsub.size() > 0 && ! overlap(inlsub) )
            {
                configs++ ;
                if ( verb)
                {
                    /* first the host */
                    for (int r=0 ; r < rows ; r++)
                    for (int c=0 ; c < cols ; c++)
                    {
                        if ( bits[r][c] > 0 )
                            System.out.print("(" + r + " " + c + ")") ;
                    }
                    System.out.println() ;
                    /* then the list of k-polys ina single row */
                    /* print a format suitable for Til.java
                    */
                    for( int n= 0 ; n < inlsub.size() ; n++)
                    {
                        FreePoly kpol = inlsub.elementAt(n) ;
                        for (int r=0 ; r < kpol.rows ; r++)
                        for (int c=0 ; c < kpol.cols ; c++)
                        {
                            if ( kpol.bits[r][c] > 0 )
                                System.out.print("(" + r + " " + c + ")") ;
                        }
                        if ( n == 0 || n < inlsub.size()-1 )
                            System.out.print(" | ") ;
                    }
                    System.out.println("") ;
                }
            }
        }
        return configs ;
    }


    /** 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 FreePolySet.java

```
/** @file
* The set of n-ominoes with a r X c bounding box and fixed n.
* @author R. J. Mathar
*/

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

/**
* @brief compute the set of all free n-ominoes with given bounding rectangle.
* @since 2019-05-11
*/
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 ;

/**
* 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.
*/
public void create()
{
    /* no solution if there are more n than r*c
    */
    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
    */
    Composit rowComp = new Composit(n,rows,1,cols) ;

    /* accumulate only once all possible bit sets of the rows
    */
    Vector<Composit> bitsets = new Vector<Composit>() ;
    for (int bweit =0 ; bweit <= cols ; bweit++)
    {
        Composit hamm = new Composit(bweit,cols,0,1) ;
        bitsets.add(hamm) ;
    }


    /* loop over all compoistions of the row sums */
    for ( int[] rc : rowComp.comps)
    {
        /* skip those where the reverse of the composition would be smaller,
        * because we'll create them anyway by the 180 deg rotations...
        */
        int[] rcrev = Composit.reverse(rc) ;
        if ( Composit.compareTo(rcrev,rc) >= 0  )
        {
            /* now row sums are fixed ; distribute them over
            * rows: need binary vectors with rc[] bits set.
            */
            byte[][] bits = new byte[rows][cols] ;
            create(bits,0,rc,bitsets) ;
        }
    }
} /* create */

/** 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.
* @param bitsets bitsets[b] contains all ways to distribute b bits over columns.
*    The parameter may be null, then vector is inefficiently recomputed locally.
*/
public void create(byte[][] bits, int prow, int [] rc,
    final Vector<Composit> bitsets)
{
    /* 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. Check each of them
    * in turn if that has a 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 reuirement) satisfied ?
        */
        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.
            */
            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] == (byte) 1)
                percol = true ;

        if ( percol)
        {
            for(int c =0 ; c < cols ; c++)
                bits[prow][c] = (byte) brow[c] ;

            if ( prow == rows-1)
                /* reached a leave of the search scan
                */
                create(bits) ;
            else
            {
                /* recursively add another row */
                create(bits, prow+1,rc,bitsets) ;
            }
        }
    }
```

```
} /* create */

/** Check whether bits[][] is a valid n=ominoe and
* add to the list if not yet present.
* @param bits
*/
void create(byte[][] bits)
{
    /* this is the set of squares that are not yet associated
    * with the cluster.
    * Check that all parts of the composition of the column sums are >0
    */
    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 ;
    }
    Vector<byte[]> freeSet = new Vector<byte[]>() ;
    for(int r=0 ; r < rows ; r++)
    for(int c=0 ; c < cols ; c++)
        if ( bits[r][c] > 0 )
        {
            byte[] coo= new byte[2] ;
            coo[0] = (byte) r ;
            coo[1] = (byte) c ;
            freeSet.add(coo) ;
        }

    /* this set contains [r][c] lists of 2d cordinates
    * of set bits (squares of the n-ominoe) connected
    * with the first square. If all conncetions are checked,
    * the size of this vector must be n if the n-ominoe is connected
    */
    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
        */
        boolean enlarged = false ;
        for( byte[] cand: freeSet)
        {
            /* is this candidate neighbout of any in the conecluster?
            */
            boolean isne = false ;
            for( byte[] inclus : coneCluster)
            {
                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 */
                coneCluster.add(cand) ;
                freeSet.remove(cand) ;
                enlarged = true ;
                break ; /* needed to avoid scanning the mof */
            }
        }
```

```
        if ( ! enlarged)
            break ;
    }

    if ( coneCluster.size() == n)
    {
        FreePoly cand =new FreePoly(bits) ;
        /* check wheter this is a new n-omino */
        boolean known =false ;
        for( FreePoly pol :  polys)
            if ( FreePoly.compareTo(pol, cand) == 0 )
            {
                known = true ; break;
            }

        /* append the new polyomino if it differs from all the known ones.
        */
        if ( ! known)
            polys.add(cand) ;
    }
} /* 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 */

/** Main program
* usage: java -cp . FreePolySet [-v] #size #straight
*/
public static void main(String[] args)
{
    /* if verb=true, print also the {0,1} matrices
    */
    boolean verb = false ;

    for( int optind =0 ; optind < args.length ; optind++)
    {
        if ( args[optind].equals("-v") )
            verb =true ;
    }

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

    /* last command line argument is the straight polyomino size
    */
    int n = Integer.parseInt(args[args.length-2]) ;

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

    /* loop over all numbers of columns
    */
    for(int c= 1; c<=n; c++)
    {
```

```
            /* need r*c >= n, so don't start at 1..
            */
            int rmin = Math.max(n/c,c) ;
            for(int r= rmin ; r+c-1 <=n ; r++)
            {
                FreePolySet po = new FreePolySet(n,r,c) ;
                po.create() ;
                /* loop over all free n-ominoes
                */
                for( FreePoly pol : po.polys)
                {
                    /* count al ways of filling with k-omines */
                    tot += pol.fill(k,verb) ;
                }
            }
        }
        System.out.println("# n= "+ n + " k= " + k + " : " + tot) ;
    } /* main */
} /* FreePolySet */
```

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

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