Square Packing

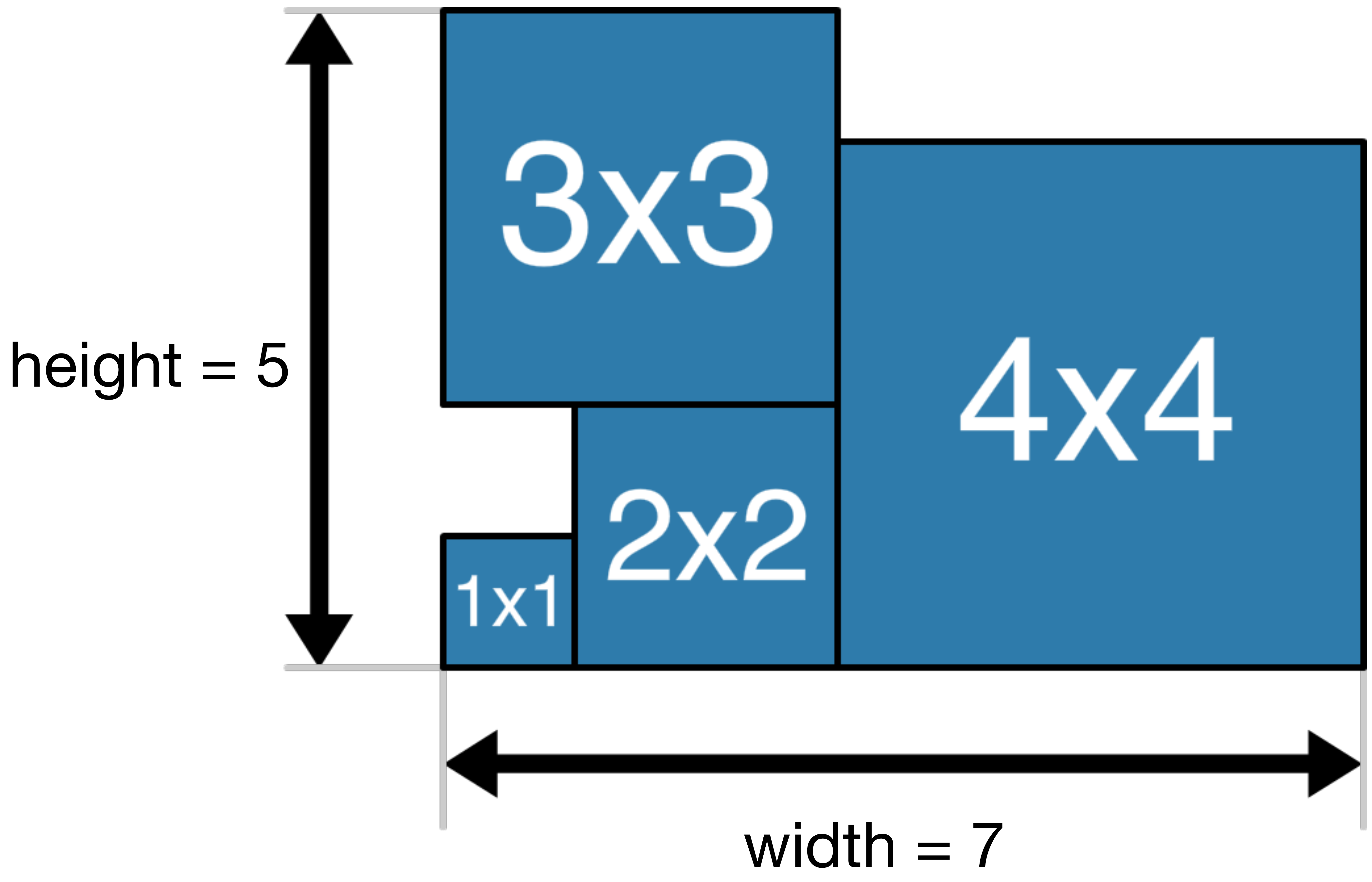
Peter Stuckey & Guido Tack

MONASH University

Multiple Square Packing

- Given
  - k_1 1 by 1 squares
  - k_2 2 by 2 squares
  - …
  - k_n n by n squares
- Pack the squares into a rectangle of the **smallest area**

Square Packing Visualized



area = 5x7 = 35

Data and Variables

```
int: n; % number of square sizes
enum Sizes = Size(1..n);
array[Sizes] of int: ncopy;
int: maxl = sum(i in 1..n)(i*ncopy[Size(i)]);
int: mina = sum(i in 1..n)(i*i*ncopy[Size(i)]);

var n..maxl: height;
var n..maxl: width;
var mina .. n*maxl: area = height * width;

enum Squares =  Square(1..sum(ncopy));
array[Squares] of var 0..maxl: x;
array[Squares] of var 0..maxl: y;
```

Note the **tight bounds** on the variables

Determining Square Sizes

```
array[Squares] of int: size =
  array1d(Squares,
    [ i | i in 1..n, j in 1..ncopy[Size(i)]]
  );
```

For example

```
ncopy = [3,2,5,4,3]
size = [1,1,1,2,2,3,3,3,3,3,4,4,4,4,5,5,5]
```

Square Packing Constraints

- Squares fit in the rectangle

```
constraint forall(s in Squares)(
  x[s] + size[s] <= width /\
  y[s] + size[s] <= height
);
```

- Squares do not overlap

```
constraint forall(s1, s2 in Squares where s1 < s2)(
  x[s1] + size[s1] <= x[s2] \/
  x[s2] + size[s2] <= x[s1] \/
  y[s1] + size[s1] <= y[s2] \/
  y[s2] + size[s2] <= y[s1]
);
```

- Objective

```
solve minimize area;
```

Solving the Model

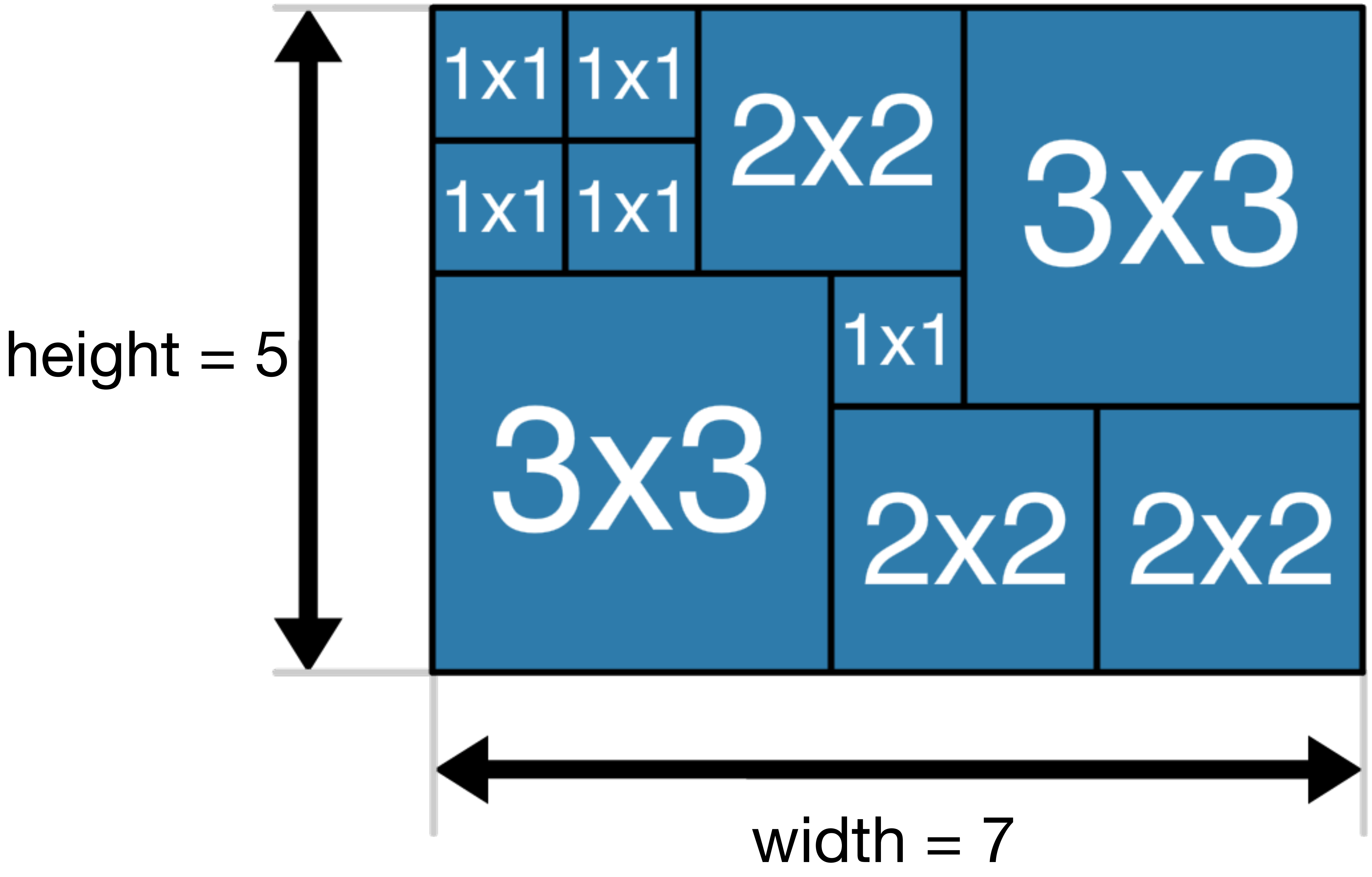
- A toy instance

```
n = 3;
ncopy = [5,3,2];
```

- Output

```
area = 35
height = 5
width = 7
x = [0, 0, 0, 0, 0, 1, 3, 5, 1, 4]
y = [0, 1, 2, 3, 4, 0, 0, 0, 2, 2]
----------
==========
Finished in 161 msec
```

A Solution to the Toy Problem



area = 5x7 = 35

Solving the Model Again

- A bigger instance

```
n = 7;
ncopy = [4,3,0,5,4,3,4];
```

- After 6s

```
area = 520
```

- After 1m

```
area = 507
```

- After 7m and up to 1h

```
area = 504
```

Improving the Model

- Global constraints
- Redundant constraints
- Symmetry breaking

The `diffn` Global Constraint

- The diffn global constraint captures exactly 2d non overlap (it should be called diff2)

```
diffn([x1, …, xn],  [y1, …, yn],
      [dx1, …, dxn],[dy1, …, dyn])
```

  ensure no two objects at positions (x_i,y_i) with dimensions (dx_i,dy_i) overlap

```
predicate diffn(array[int] of var int: x,
                array[int] of var int: y,
                array[int] of var int: dx,
                array[int] of var int: dy);
```

- Squares do not overlap

```
diffn(x, y, size, size);
```

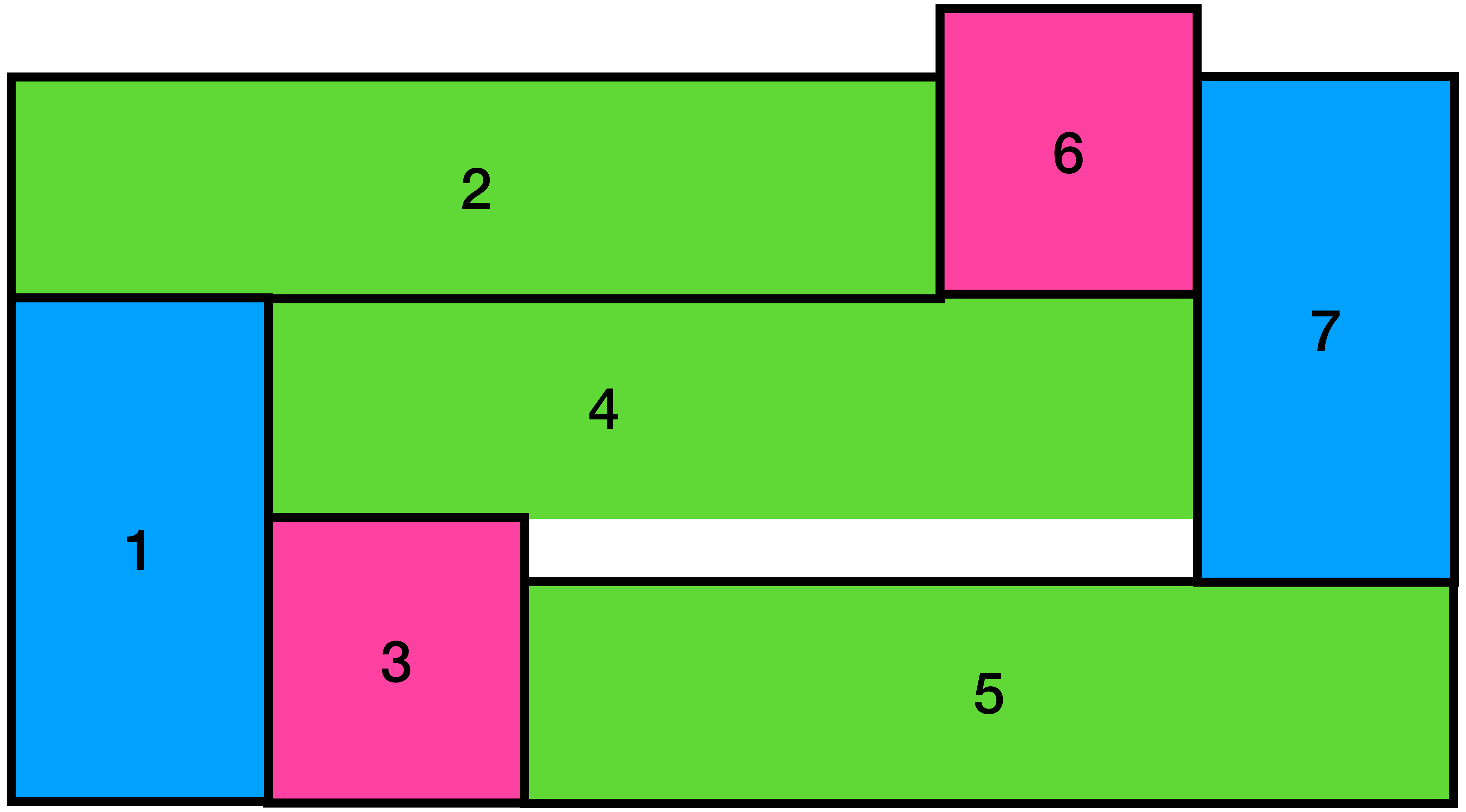
Packing and Cumulative

- If there is a packing, then the cumulative constraint must hold!
- We can add **redundant** cumulative constraints to packing problems
    - improves propagation (and hence solving)
- Squares do not overlap in the x and y dimension respectively

```
cumulative(x, size, size, height);
cumulative(y, size, size, width);
```

Packing and Cumulative

- In general cumulative constraints **do not** enforce packing, even when the the x positions are fixed



Symmetries!

- Squares of the **same size** are interchangeable, creating multiplicity of solution possibilities
- Impose an ordering on the placements of such squares
- What ordering can we use for coordinates (x, y)?
- Strict lexicographical ordering

$$(x_1, y_1) >_{lex} (x_2, y_2)$$

  - $x_1 > x_2$; or
  - if $x_1 = x_2$, then $y_1 > y_2$

The `lex_greater` Global Constraint

- The lex_greater global constraint imposes the lexicographic ordering on two *n*-tuples (encoded as arrays)

  `lex_greater([`x_1`,…,`x_n`], [`y_1`,…,`y_n`])`

  ensures that $(x_1, \ldots, x_n) >_{lex} (y_1, \ldots, y_n)$

```
predicate lex_greater(array [int] of var int: x,
                      array [int] of var int: y)
```

Ordering Squares

- The placement of a square is specified by the coordinates of its lower left hand corner

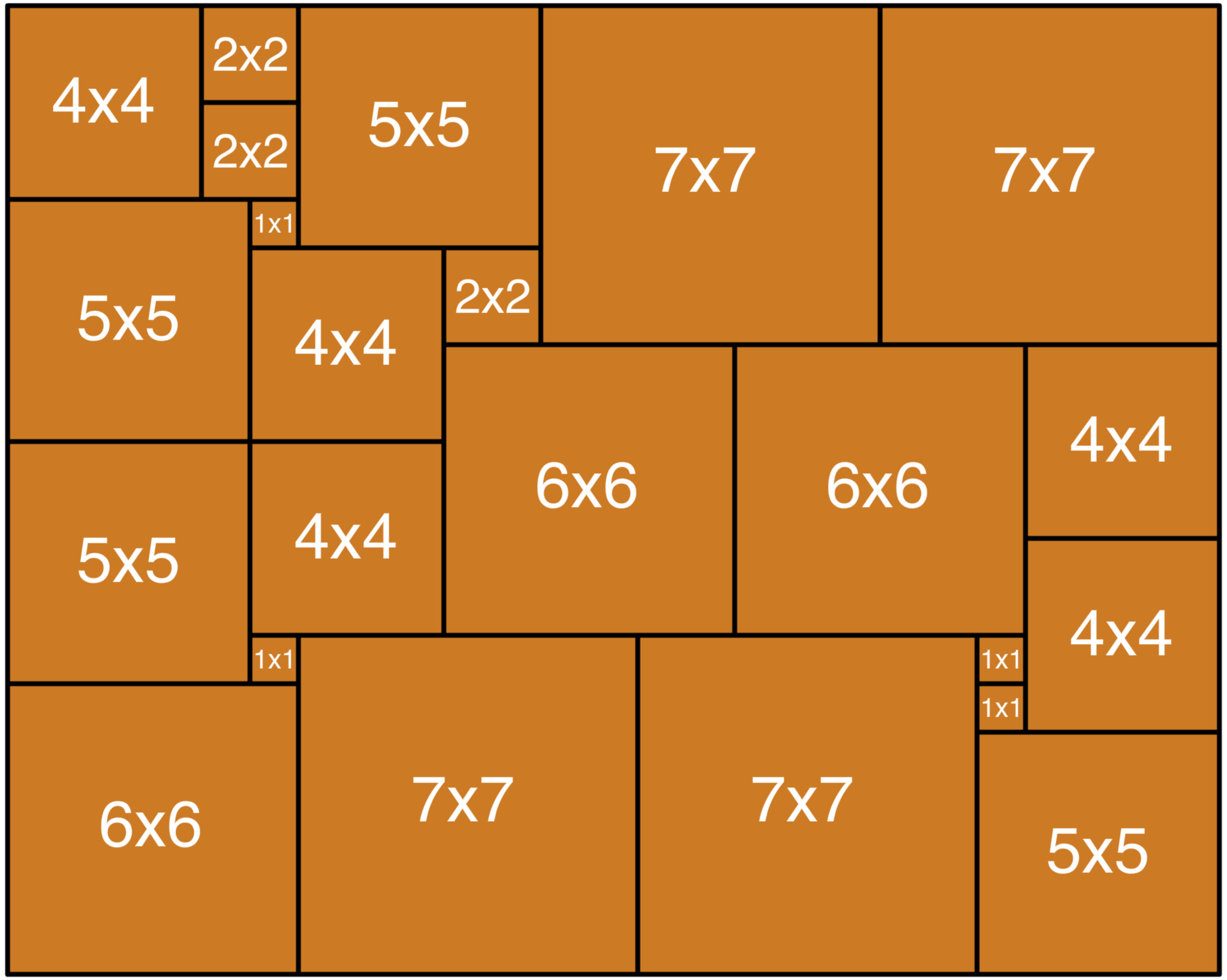
- Order squares of the same size

```
include "lex_greater.mzn";
constraint
forall(s in Squares diff {max(Squares)}
       t = enum_next(s))(
  if size[s]=size[t] then
   lex_greater(
      [x[s], y[s]],
      [x[t], y[t]]
    )
  endif
);
```

Solving the Model Again

```
area = 500
height = 20
width = 25
x = [20, 20, 5, 5, 9, 4, 4, 21, 21, 5, 5,
0, 20, 6, 0, 0, 15, 9, 0, 18, 13, 11, 6]
y = [14, 13, 13, 4, 5, 2, 0, 11, 7, 9, 5,
0, 15, 0, 9, 4, 7, 7, 14, 0, 13, 0, 13]
----------
==========
Finished in 42s 792msec
```

Solving the Model Again



Summary

- Packing problems
    - are another **common** uses of CP in the real world
    - come in lots of varieties
    - are complex discrete optimization problems
- `diffn` encodes 2D non-overlap
    - `disjunctive` encodes 1D non-overlap
- `cumulative` constraints are **redundant** for packing, but useful for improving solving

EOF