

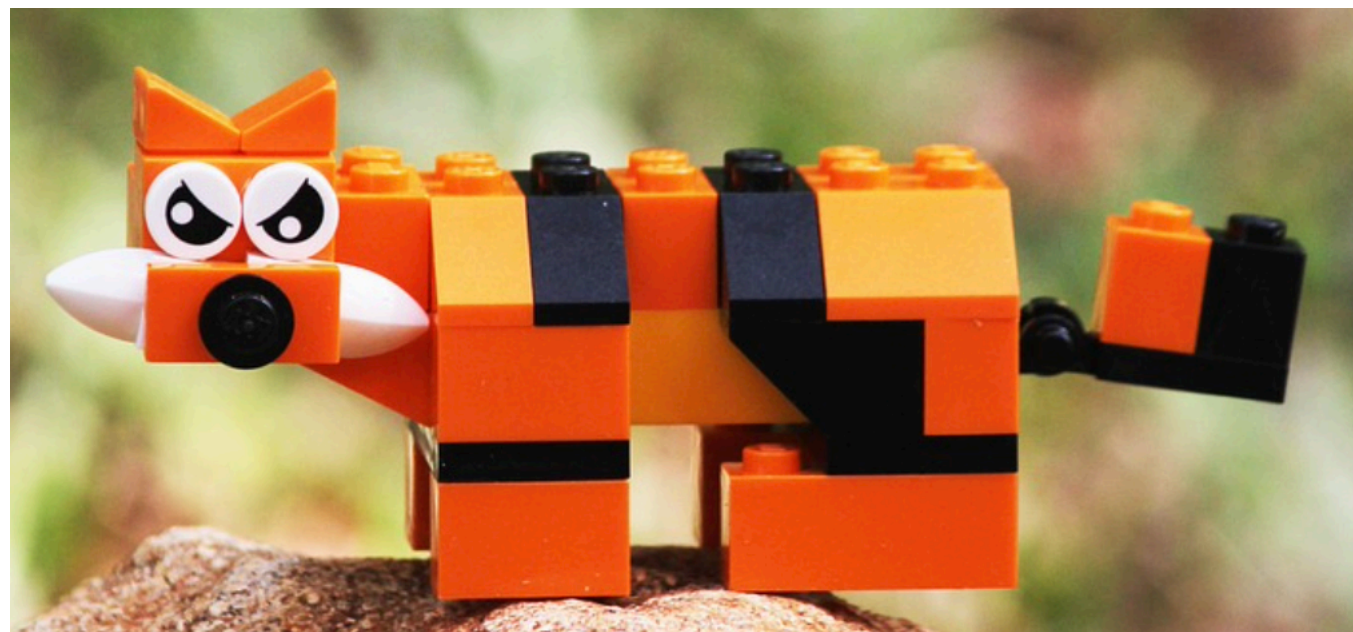
# Modeling in CP

## Bin-Packing Case Study

# Modeling is an Art

- Modeling a real world problem with variables, domains and constraints

Model

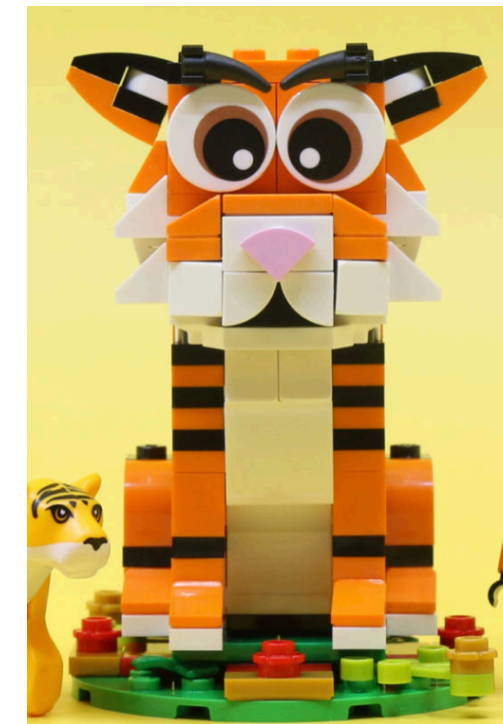
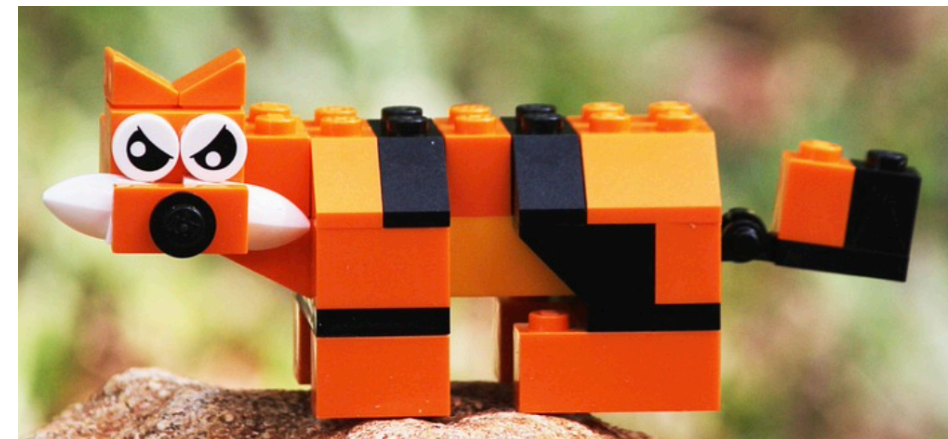


Real world problem



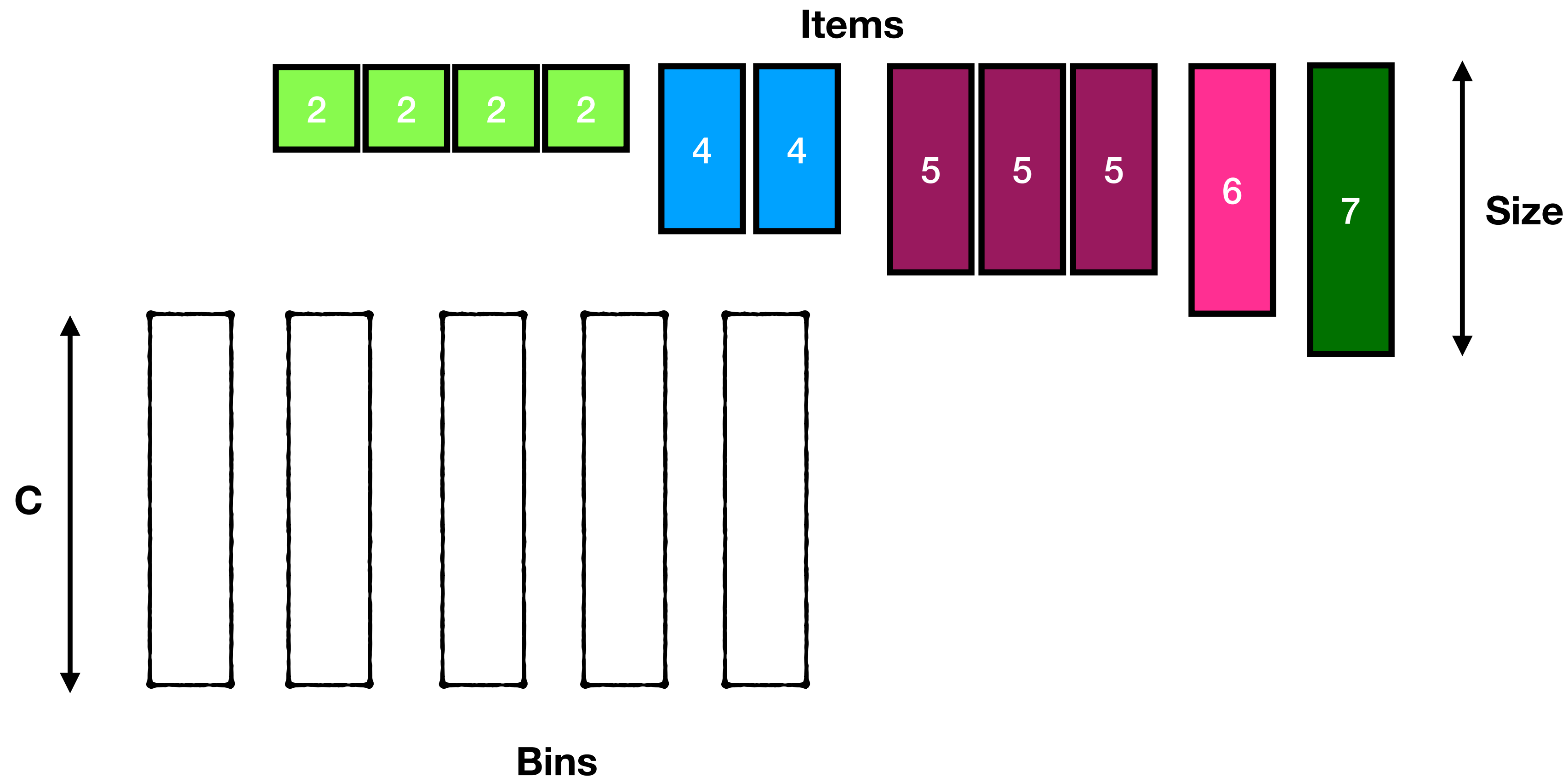
# Modeling is an Art

- ▶ For a same problem, many different models
  - Variables, domains and constraints
- ▶ The model can have dramatic effect on the solving time



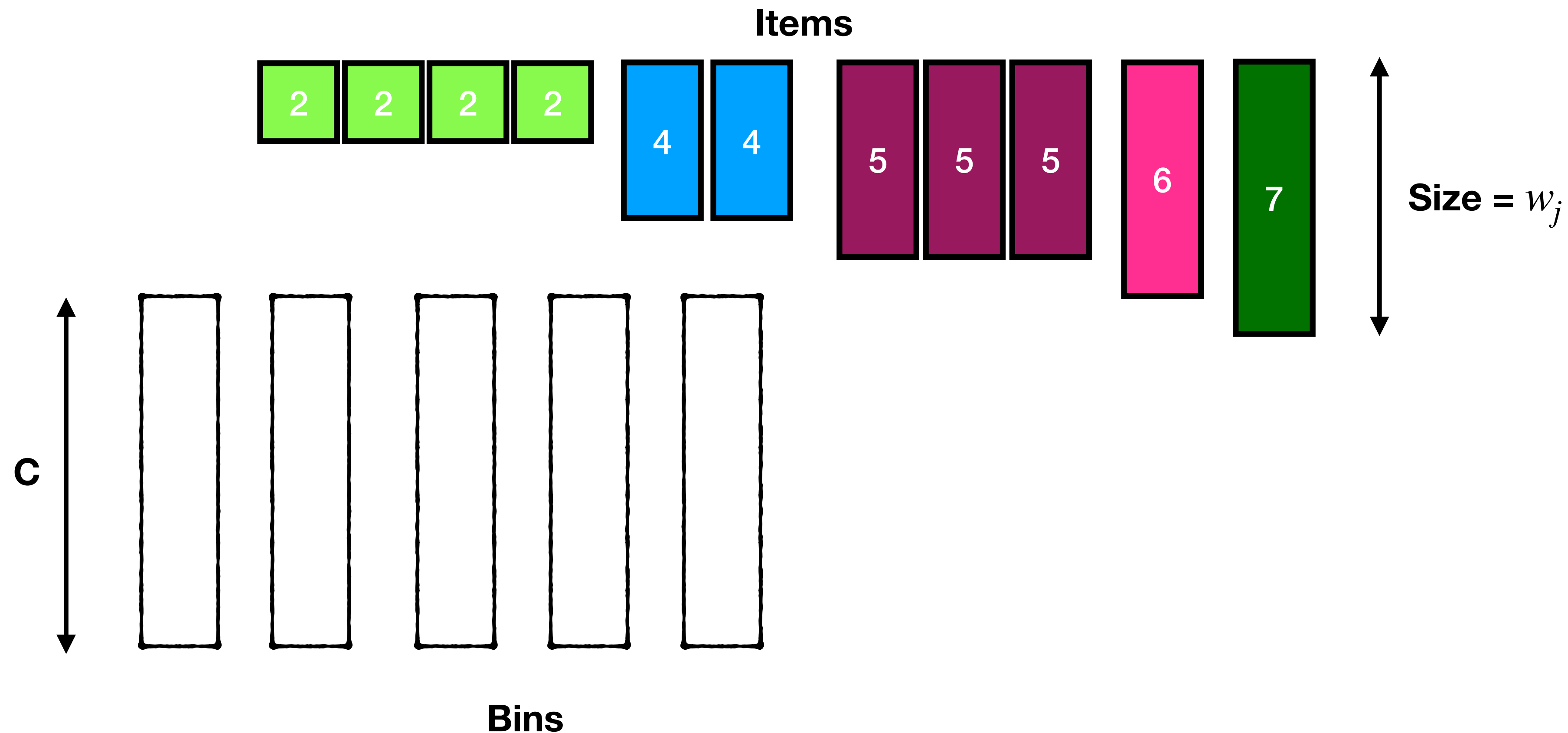
# Case Study: Bin Packing

- ▶ Given  $n$  items, the size of each item
- ▶ Given  $m$  bins, each with a same capacity  $c$
- ▶ Find a bin for each object such that the capacity of the bins is respected



# Decision variable ?

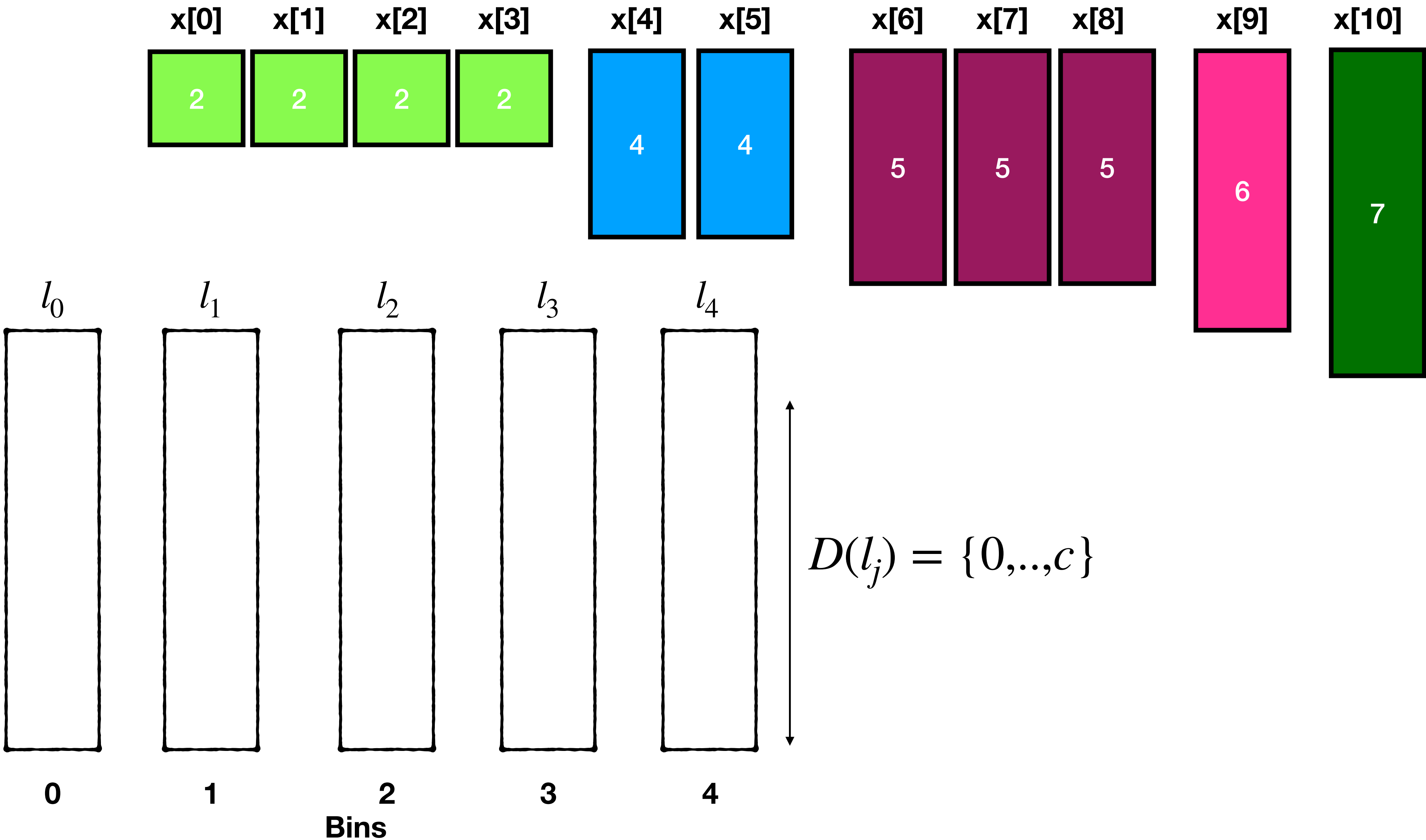
- Item point of view: in what bin do we place each item
- Bin point of view: what are the set of items allocated to each bin (set variable, more complex)



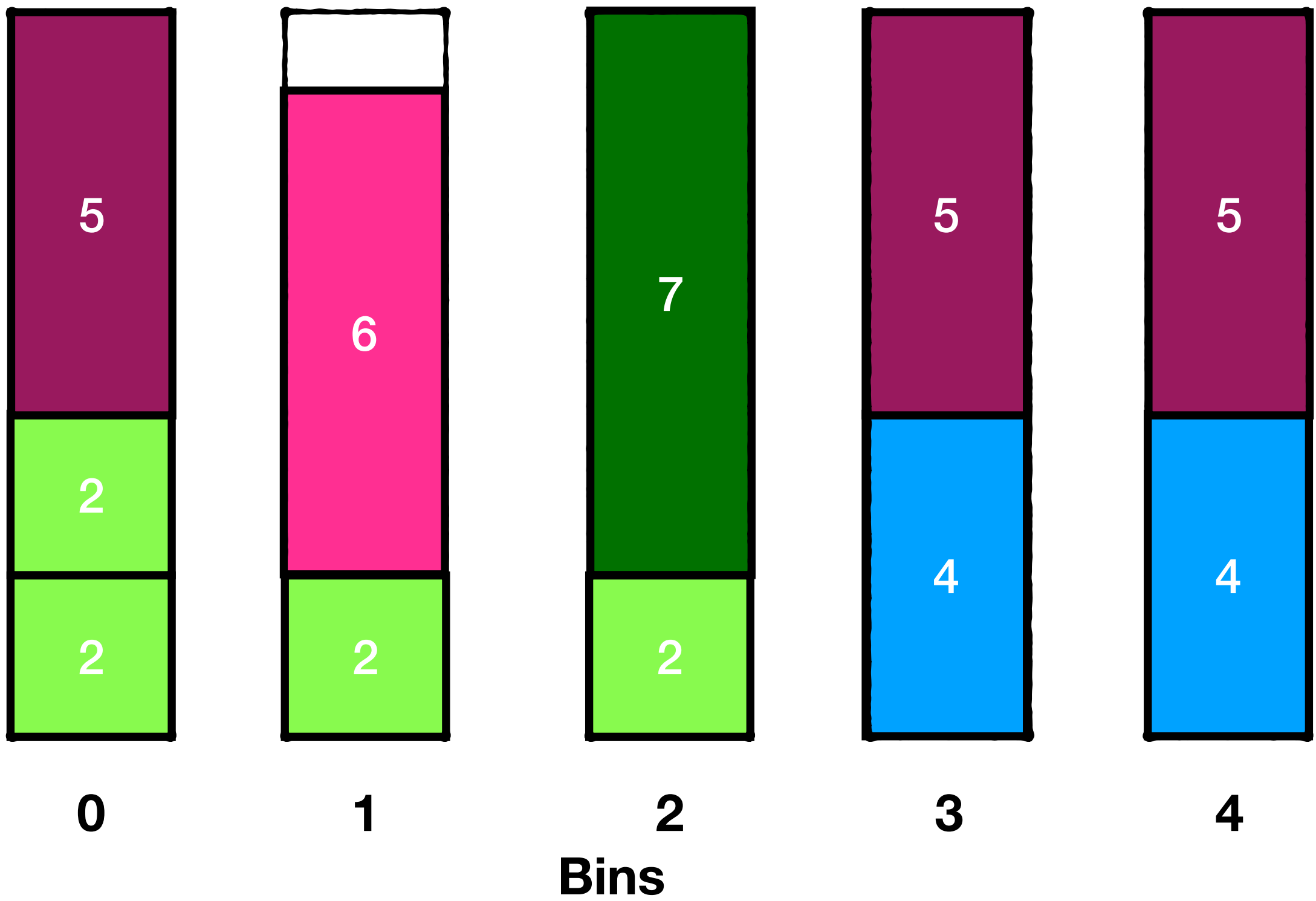
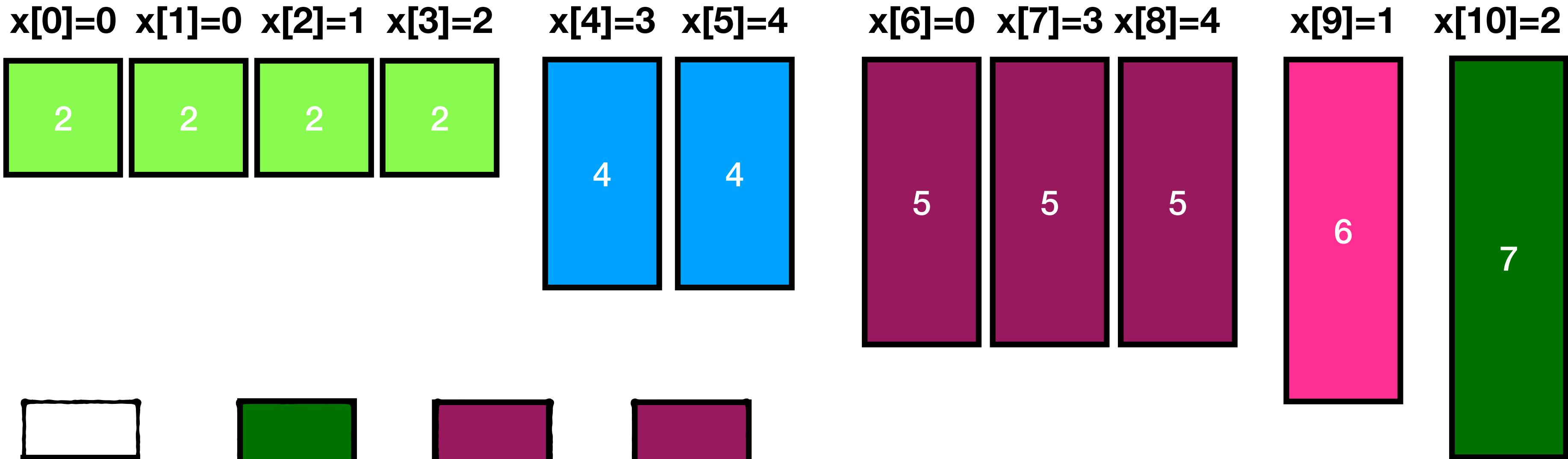
# Decision variables and Domains



$$D(x_i) = \{0,1,2,3,4\}$$

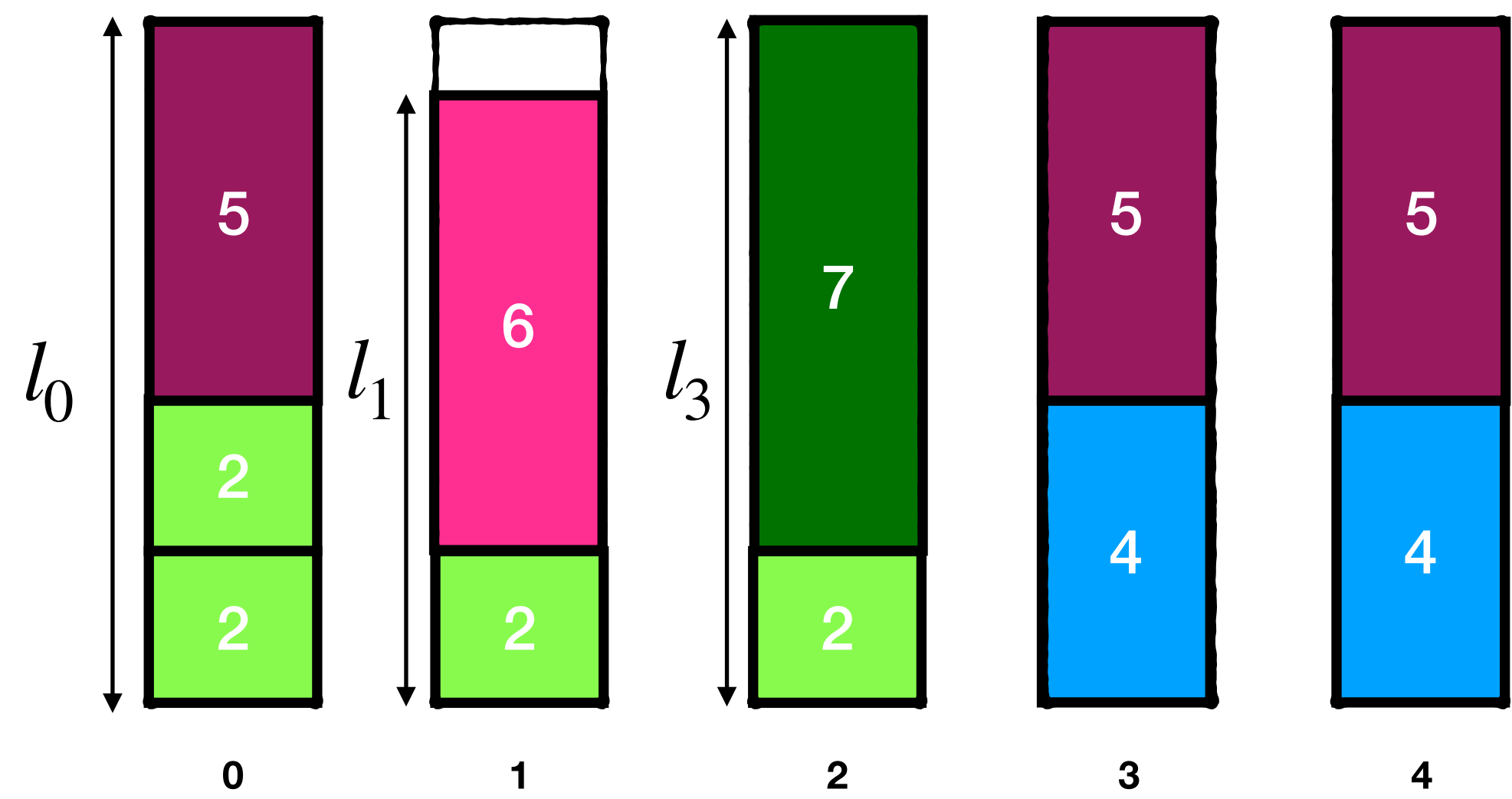


# Solution



$$\forall j \in [1..m] : l_j = \sum_{i \in [1..n]} (x_i = j) \cdot w_i$$

$$D(l_j) = \{0, \dots, c\}$$



# Bin-Packing Model

```

int capa = 9;
int [] items = new int[] {2,2,2,2,4,4,5,5,5,6,7};

int nBins = 5;
int nItems = items.length;

Solver cp = makeSolver();
IntVar [] x = makeIntVarArray(cp, nItems,nBins);
IntVar [] l = makeIntVarArray(cp, nBins, capa+1);

BoolVar [][] inBin = new BoolVar[nBins][nItems]; // inBin[j][i] = 1 if item i is placed in bin
j
// bin packing constraint
for (int j = 0; j < nBins; j++) {
    for (int i = 0; i < nItems; i++) {
        inBin[j][i] = isEqual(x[i], j);
    }
}
for (int j = 0; j < nBins; j++) {
    IntVar[] wj = new IntVar[nItems];
    for (int i = 0; i < nItems; i++) {
        wj[i] = mul(inBin[j][i], items[i]);
    }
    cp.post(sum(wj, l[j]));
}

```

$$l_j = \sum_{i \in [1..n]} (x_i = j) \cdot w_i$$

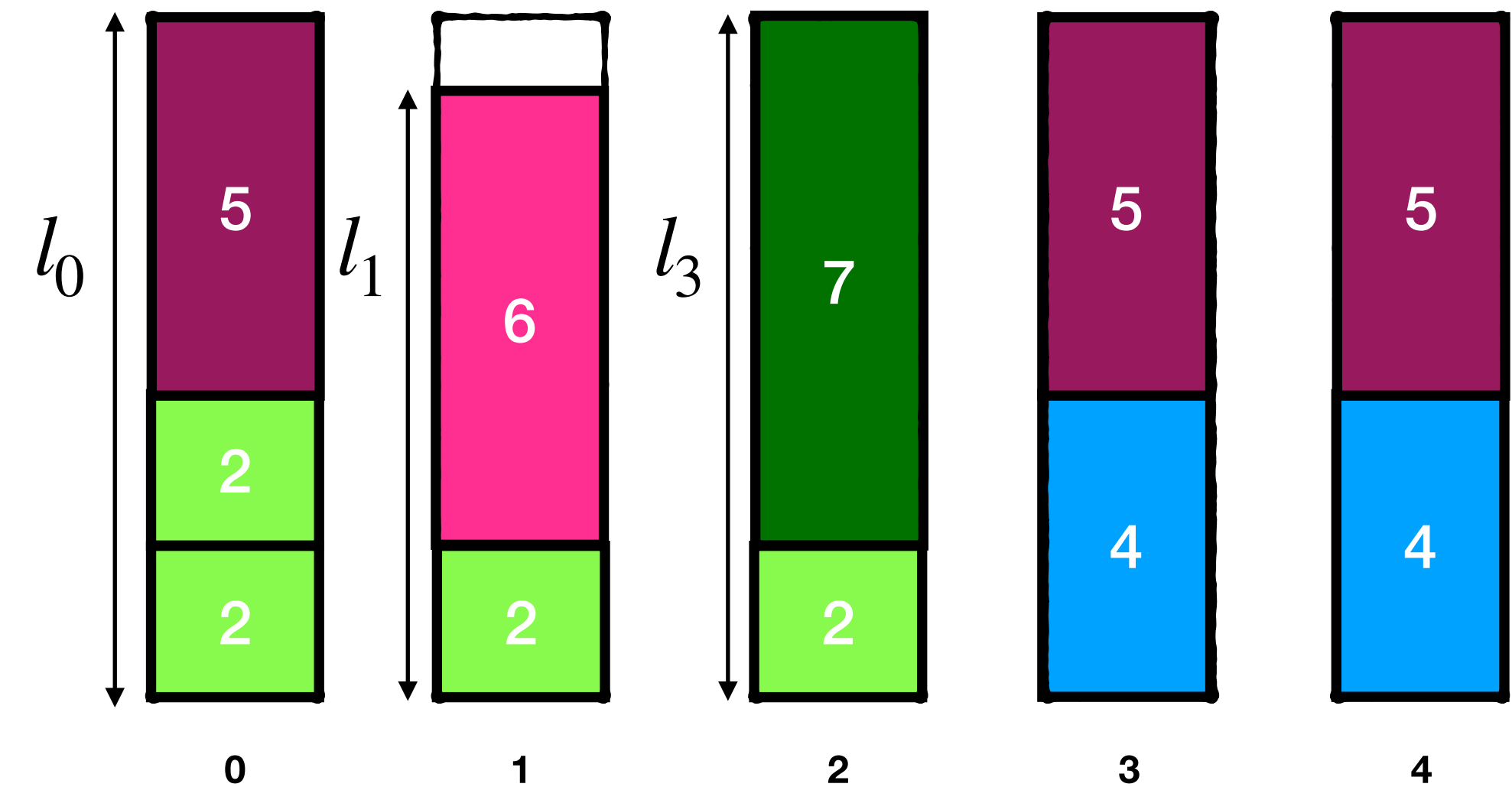
```

DFSearch dfs = makeDfs(cp, firstFail(x));

```

# Global Constraints for Bin-Packing

$$\forall j \in [1..m] : l_j = \sum_{i \in [1..n]} (x_i = j) \cdot w_i$$



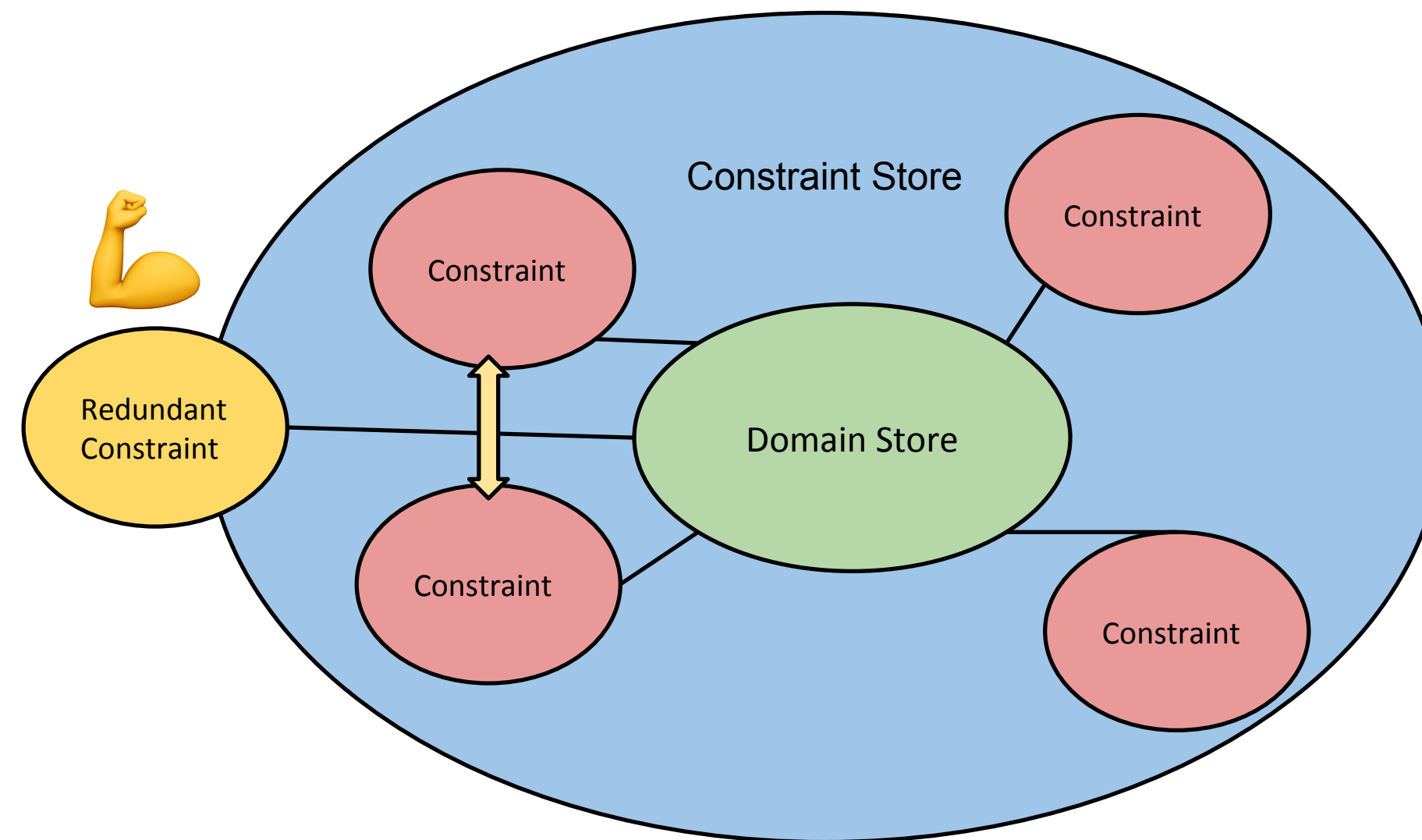
- This kind of constraint is very frequent, most of the solvers call it  $\text{BinPacking}([l_1, \dots, l_m], [x_1, \dots, x_n], [w_1, \dots, w_n])$



- Shaw, Paul. "A constraint for bin packing." CP 2004.
- Schaus, Pierre. "Solving balancing and bin-packing problems with constraint programming." *PhD Thesis* (2009)

# Redundant Constraints

- Redundant Constraints:
  - Do not exclude any previous solution
  - Improve the pruning of the search space (better communication between constraints)



# How to find redundant constraints for your model ?

- Express properties of the solution
- Derive consequences of (combinations of) constraints
- BinPacking:

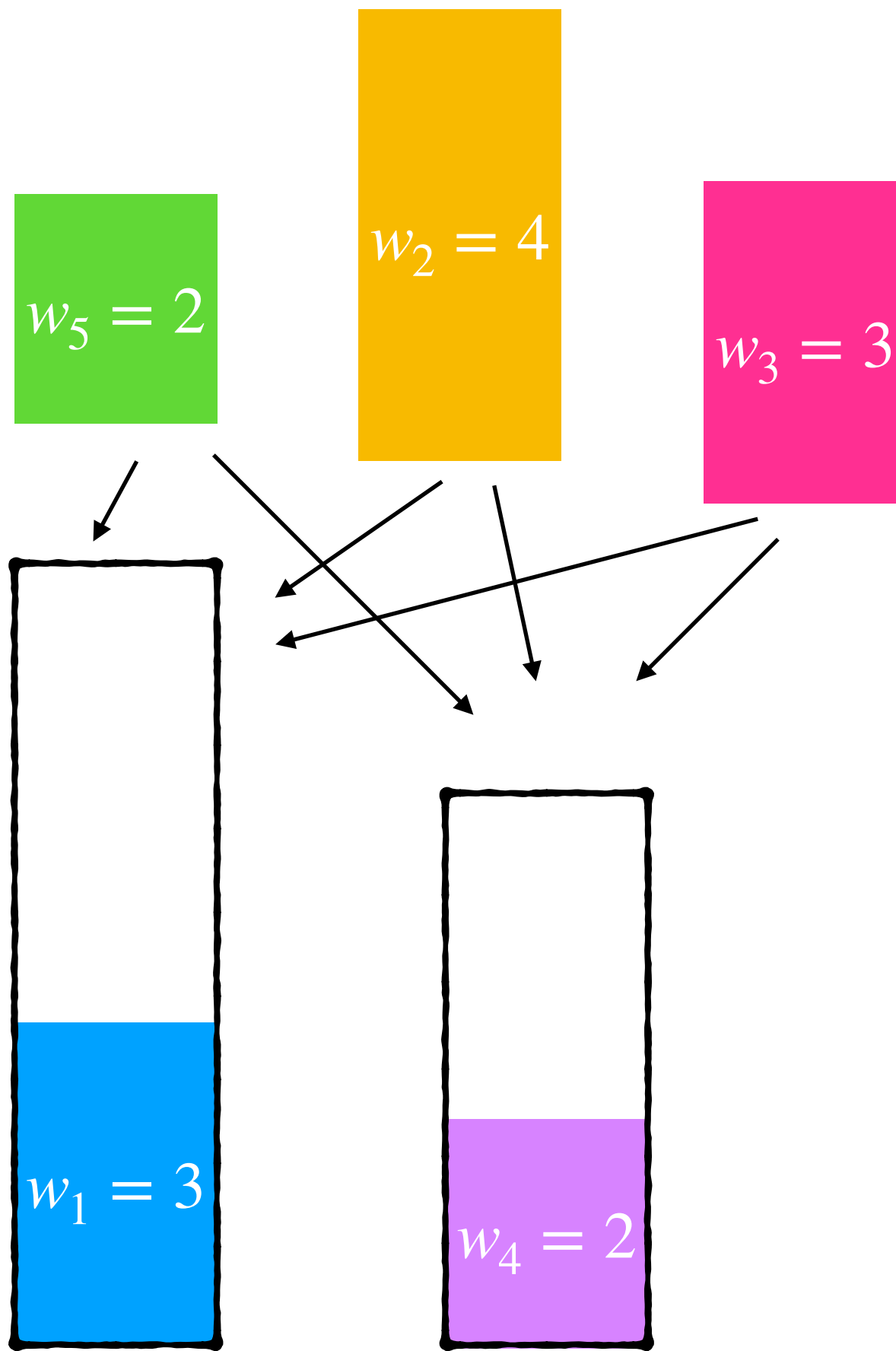
$$\forall j \in [1..m] : l_j = \sum_{i \in [1..n]} (x_i = j) \cdot w_i$$

$$\sum_{j \in [1..m]} l_j = \sum_{i \in [1..n]} w_i$$



# Redundant Constraint

Infeasible, but not detected by  $\forall j \in [1..m] : l_j = \sum_{i \in [1..n]} (x_i = j) \cdot w_i$



Failure detected by redundant constraints

$$\sum_{j \in [1..m]} l_j = \sum_{i \in [1..n]} w_i$$

$$[3..7] + [2..5] = 14$$

$$[5..12] = 14$$

# Bin-Packing Model



```
int capa = 9;
int [] items = new int[] {2,2,2,2,4,4,5,5,5,6,7};

int nBins = 5;
int nItems = items.length;

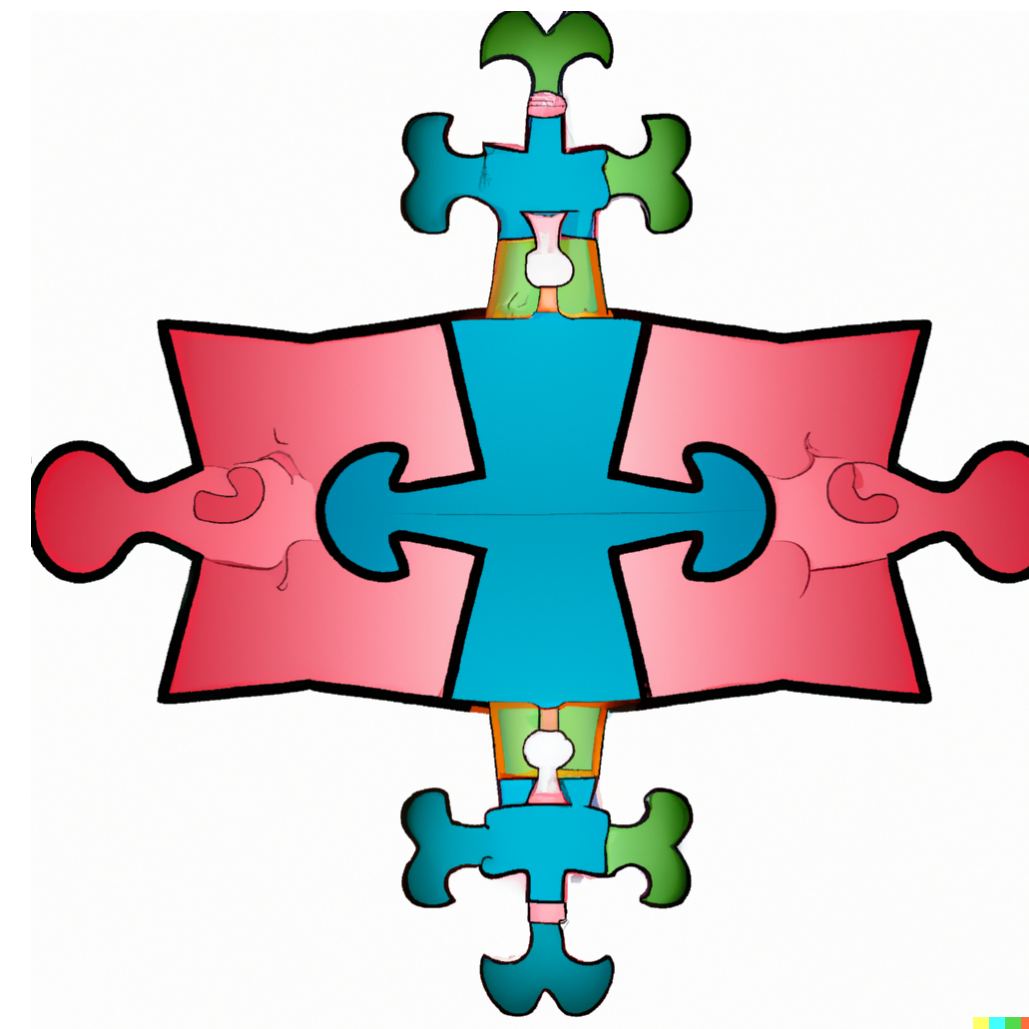
Solver cp = makeSolver();
IntVar [] x = makeIntVarArray(cp, nItems,nBins);
IntVar [] l = makeIntVarArray(cp, nBins,capa+1);

BoolVar [][] inBin = new BoolVar[nBins][nItems]; // inBin[j][i] = 1 if item i is placed in bin j
// bin packing constraint
for (int j = 0; j < nBins; j++) {
    for (int i = 0; i < nItems; i++) {
        inBin[j][i] = isEqual(x[i], j);
    }
}
for (int j = 0; j < nBins; j++) {
    IntVar[] wj = new IntVar[nItems];
    for (int i = 0; i < nItems; i++) {
        wj[i] = mul(inBin[j][i], items[i]);
    }
    cp.post(sum(wj, l[j]));
}

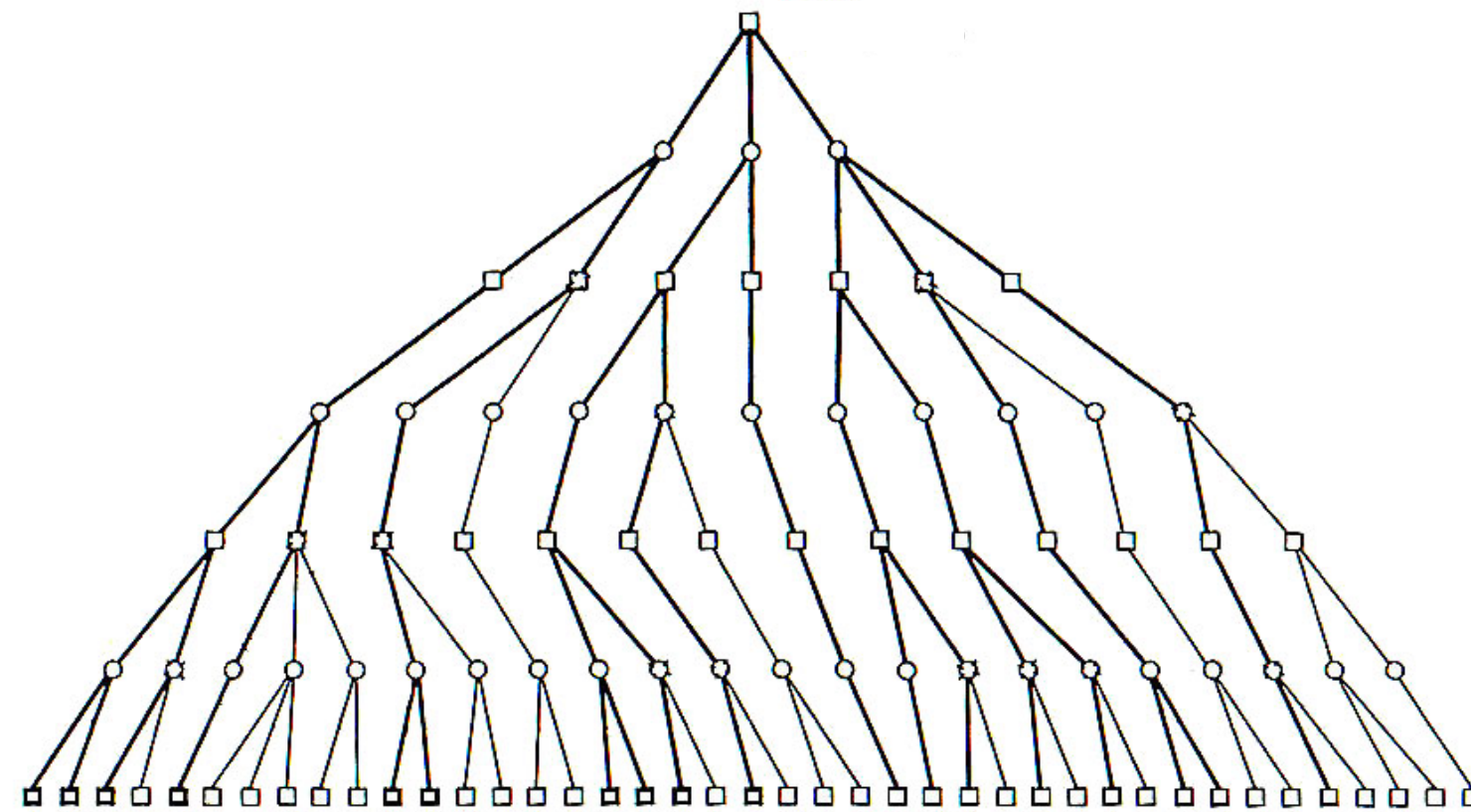
// redundant constraint 💪
cp.post(sum(l, IntStream.of(items).sum()));
```

# Bin-Packing

Symmetries



- ▶ Many problems naturally exhibit symmetries
- ▶ A symmetry maps solutions to solutions and non-solutions to non-solutions
- ▶ Symmetries leads to symmetrical search spaces
- ▶ Exploring symmetrical search spaces is useless
  - If no solution in one, no solution in the other



Detect and remove symmetries (dynamic or static)  
only inspect one (non-)solution in each equivalence class

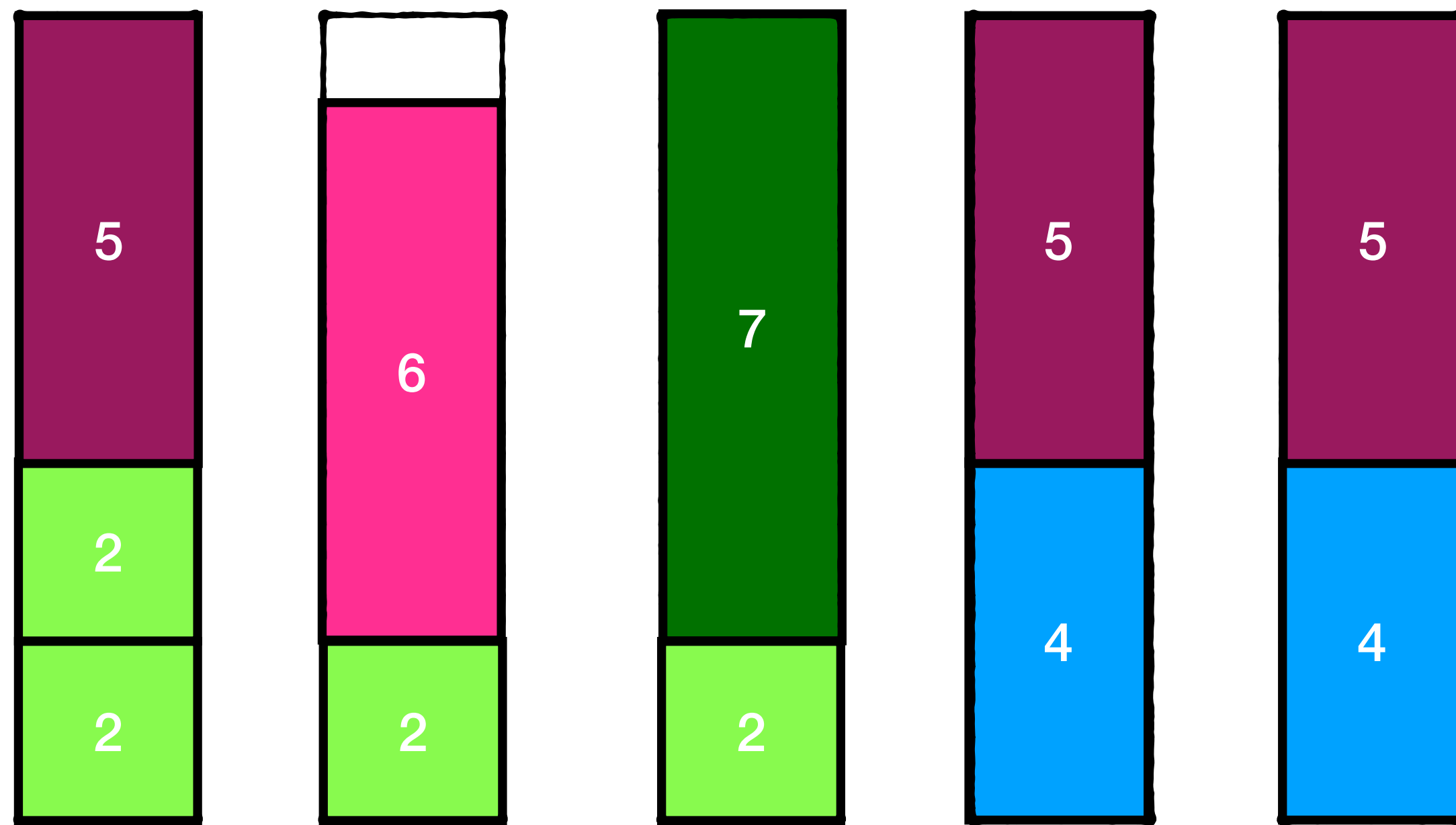
A value symmetry is a bijection  $\sigma$  on values mapping (non-)solutions to (non-)solutions:

$$\begin{array}{ccc} x_1, x_2, \dots, x_n & \xleftrightarrow{\sigma} & x_1, x_2, \dots, x_n \\ a_1, a_2, \dots, a_n & & \sigma(a_1), \sigma(a_2), \dots, \sigma(a_n) \end{array}$$

Value symmetries change the values

# Bin-Packing (value) Symmetries

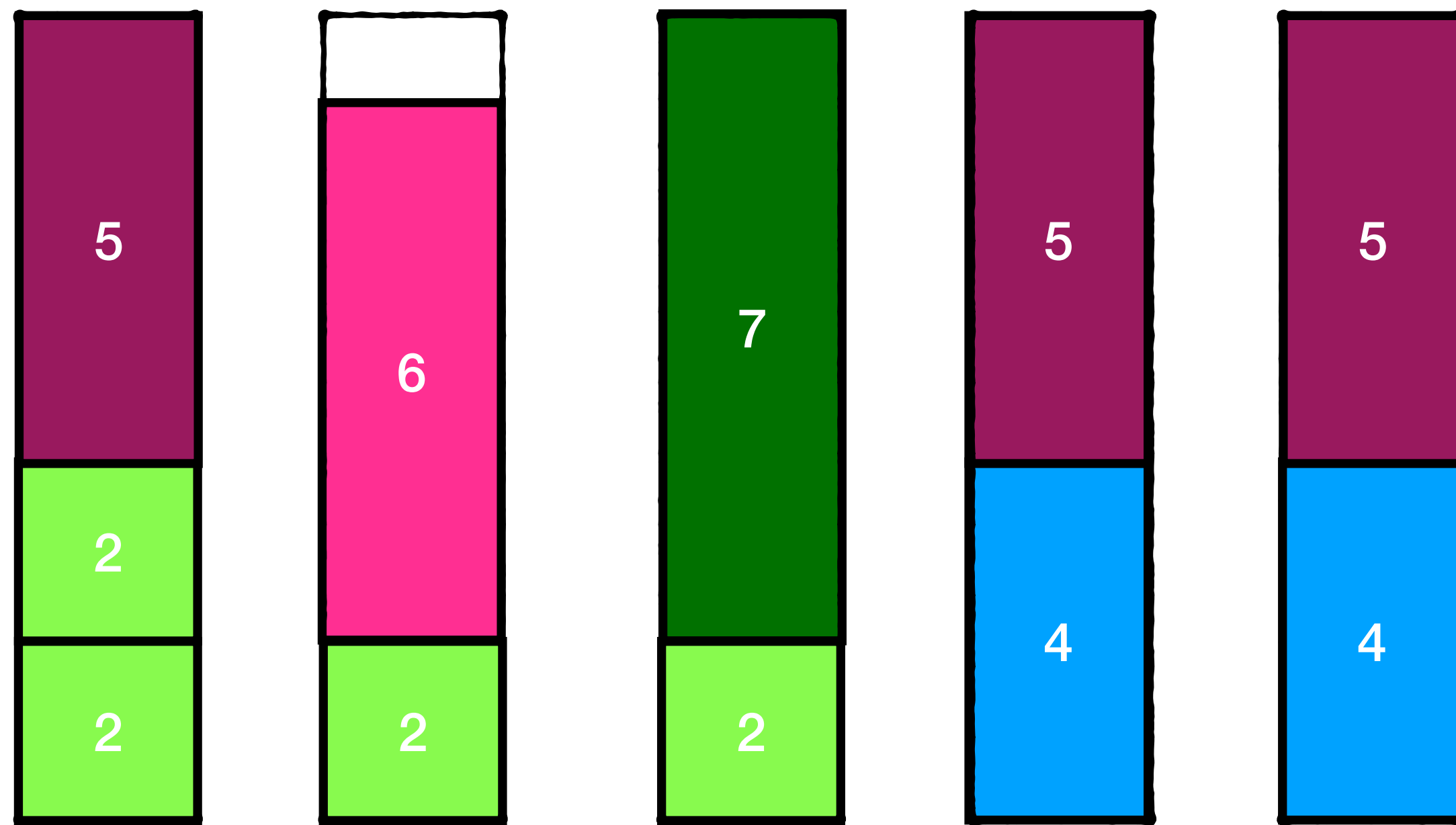
- Interchanging bins is still a valid solution



Bins

# Value symmetry breaking

- Solution: Impose an order on the bins
- For example: increasing loads  $l[0] \leq l[1] \leq l[2] \leq l[3] \leq l[4]$
- Does not remove all symmetries in case of ties



Bins

# Better: Lexicographic Constraints

- Impose a total order on bins (no ties)

```

BoolVar [][] inBin = new BoolVar[nBins][nItems]; // inBin[j][i] = 1 if item i is placed in bin j
// bin packing constraint
for (int j = 0; j < nBins; j++) {
    for (int i = 0; i < nItems; i++) {
        inBin[j][i] = isEqual(x[i], j);
    }
}
for (int j = 0; j < nBins-1; j++) {
    cp.post( inBin[j] ≤ inBin[j+1]);
}

```

Lexicographic  
Ordering

Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., & Walsh, T. (2002, September). Global constraints for lexicographic orderings. CP2002

# Variable Symmetry

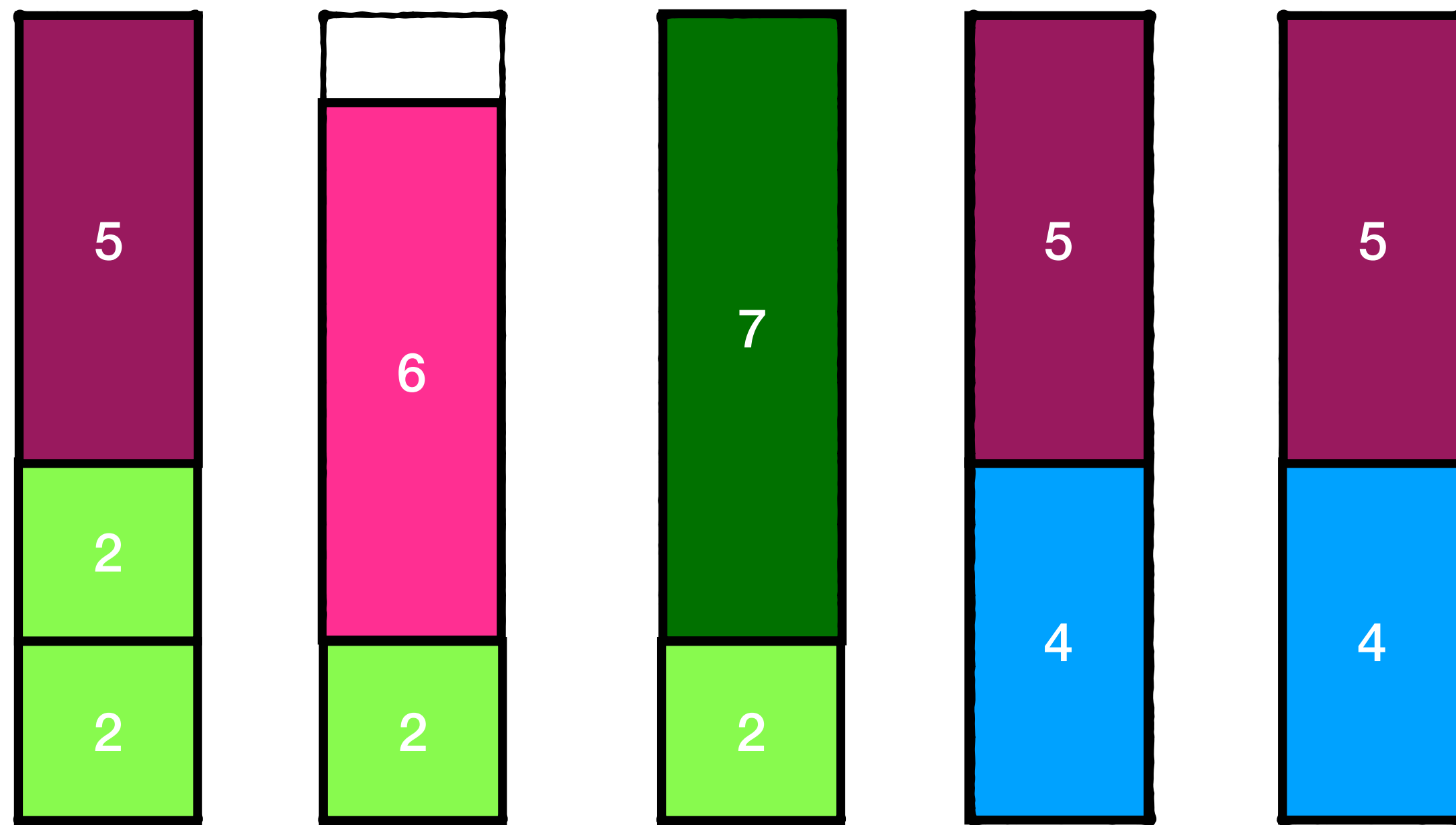
A variable symmetry is a bijection  $\sigma$  on variables mapping (non-)solutions to (non-)solutions:

$$\begin{array}{ccc} x_1, x_2, \dots, x_n & \xleftrightarrow{\sigma} & x_{\sigma(1)}, x_{\sigma(2)}, \dots, x_{\sigma(n)} \\ a_1, a_2, \dots, a_n & & a_1, a_2, \dots, a_n \end{array}$$

Variable symmetries swap the variables

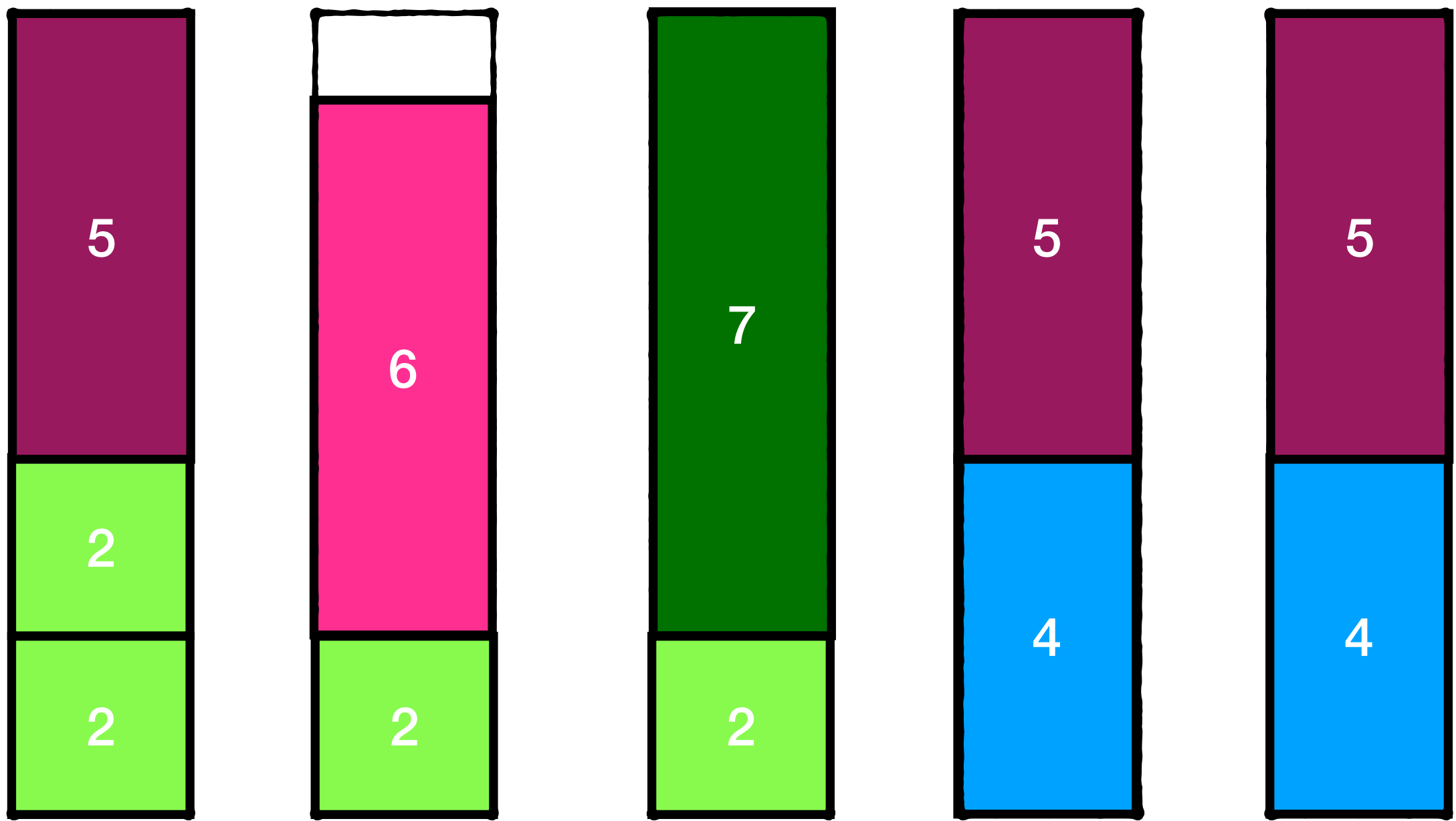
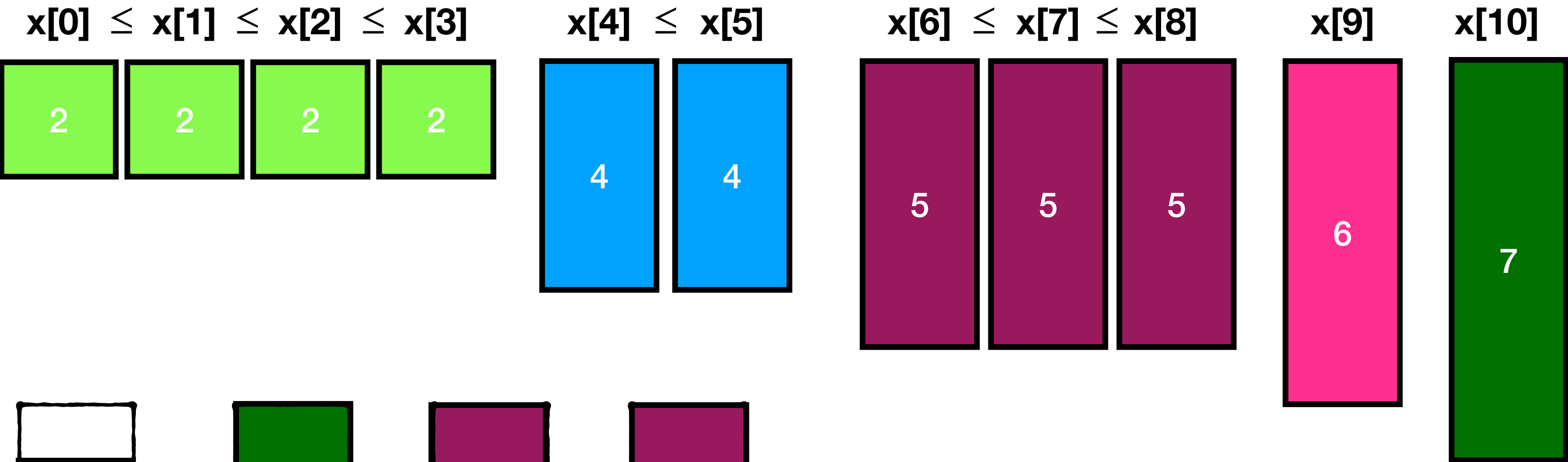
# Bin-Packing (variable) Symmetries

- Exchanging similar items



Bins

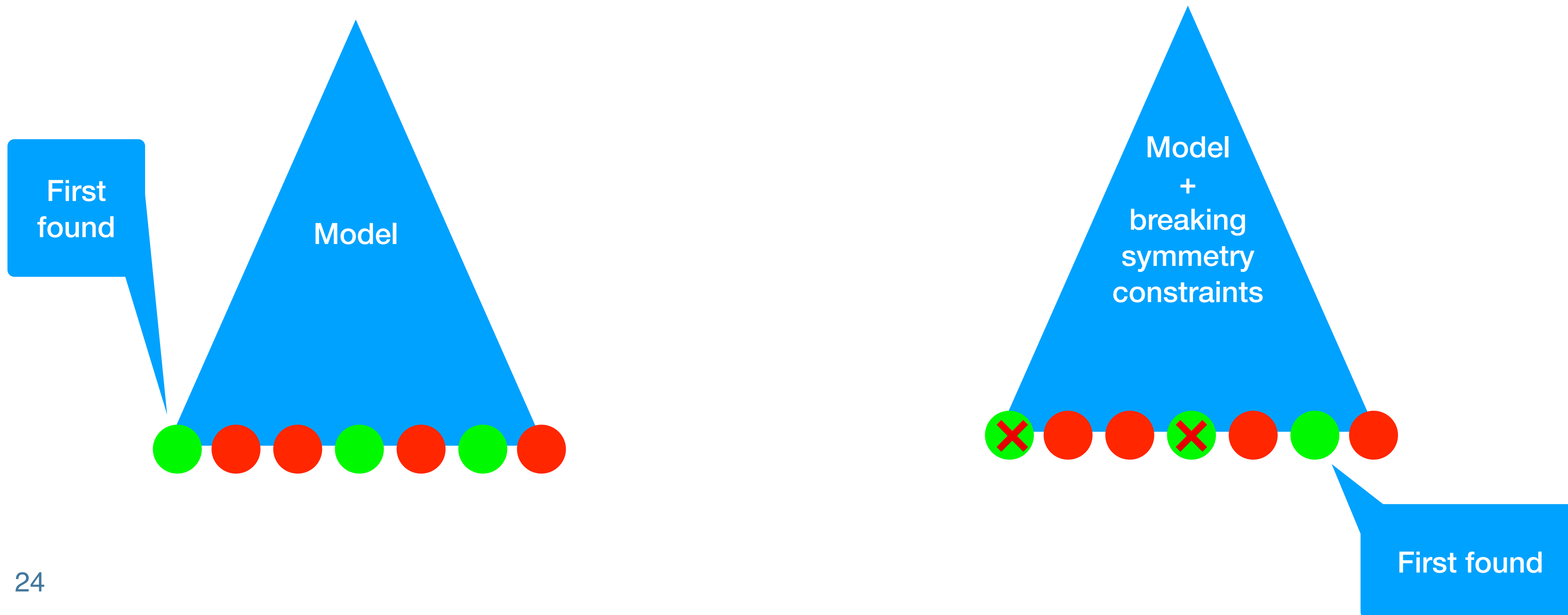
# Bin-Packing: Breaking variable symmetries



Bins

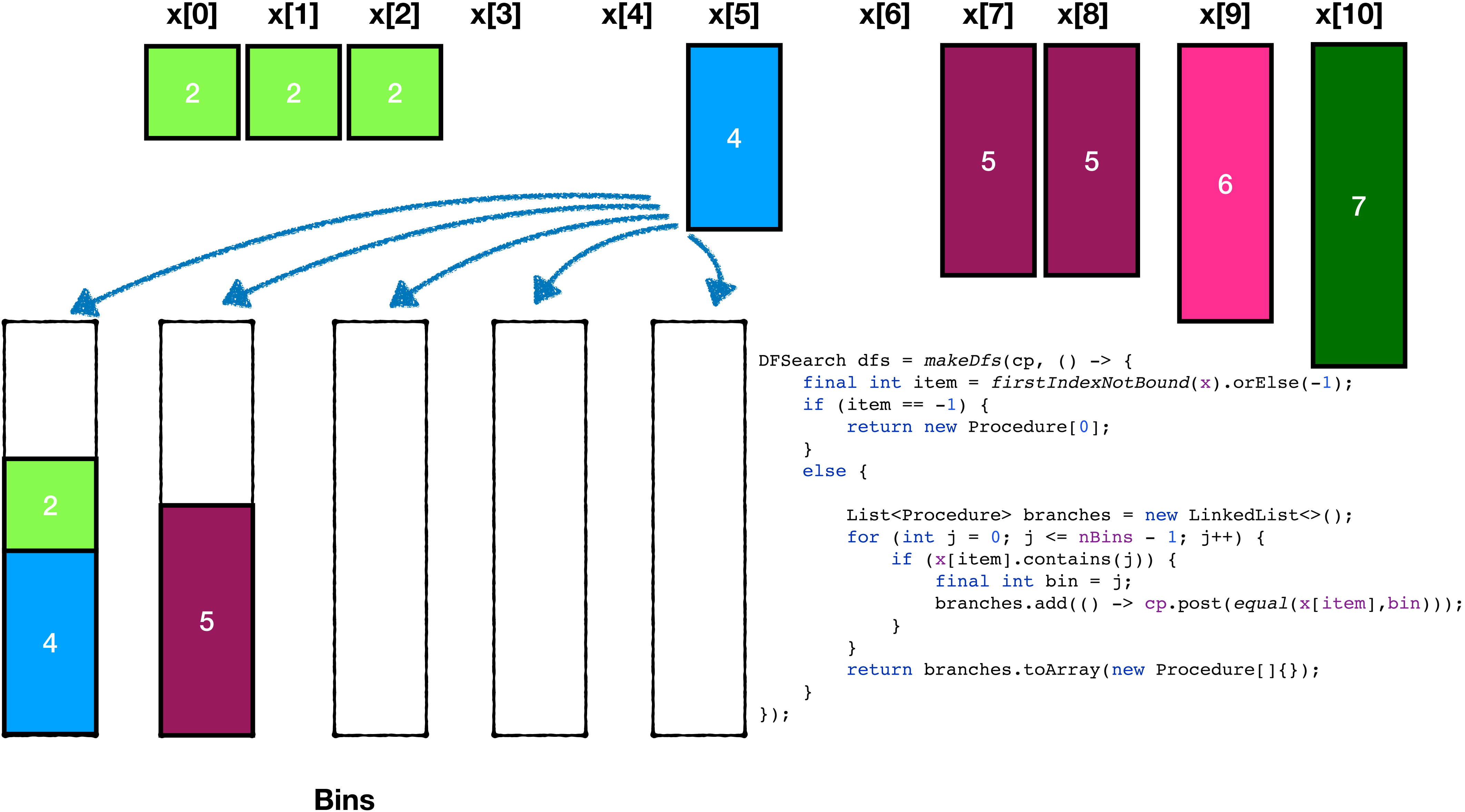
# Drawback of symmetry breaking with constraints

- ▶ ⚠ Sometimes useful, sometimes not
- ▶ Be careful because you suppress solutions.
- ▶ Consequence:
  - Solution discovered very early in the search tree might not exist anymore (bad interaction with the heuristic).

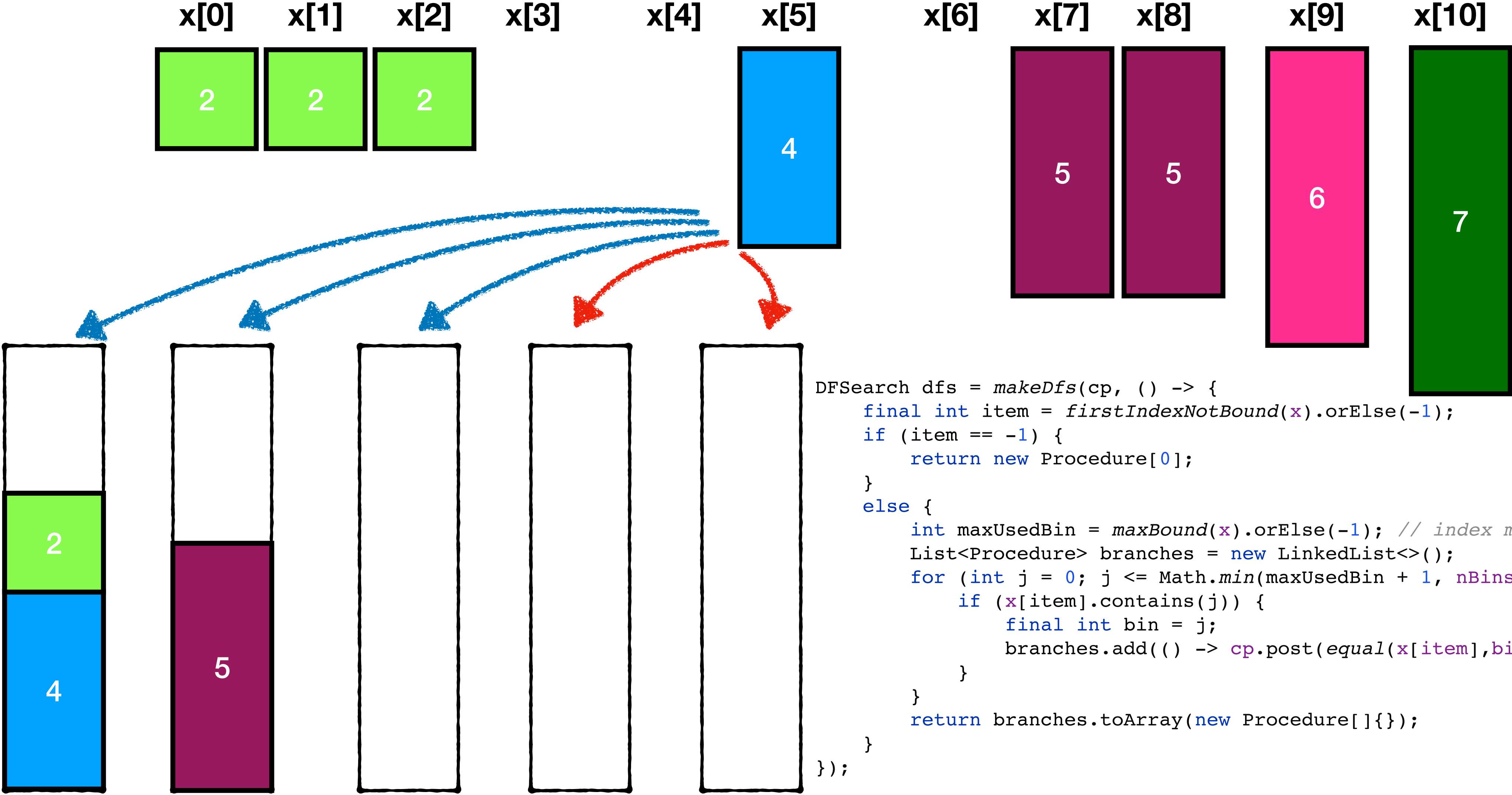


- ▶ Is it possible to remove variable/value symmetries such that the first solution remains the same ?
- ▶ Yes! Dynamic symmetry breaking = Add constraints **during search**
  - each time a (non-)solution is found)
  - Special search heuristic

# Dynamic Symmetry Breaking during Search



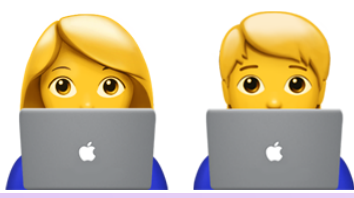
# Dynamic Symmetry Breaking during Search



```
DFSearh dfs = makeDfs(cp, () -> {
    final int item = firstIndexNotBound(x).orElse(-1);
    if (item == -1) {
        return new Procedure[0];
    }
    else {
        int maxUsedBin = maxBound(x).orElse(-1); // index max used bin
        List<Procedure> branches = new LinkedList<>();
        for (int j = 0; j <= Math.min(maxUsedBin + 1, nBins - 1); j++) {
            if (x[item].contains(j)) {
                final int bin = j;
                branches.add(() -> cp.post(equal(x[item], bin)));
            }
        }
        return branches.toArray(new Procedure[ ]{});
    }
});
```

- ▶ Static Symmetry Breaking
  - Use different variables
  - Add constraints to the model (ex: lexicographic)
- ▶ Dynamic Symmetry breaking (during search)
  - Add constraints during the search (each time a (non—solution is found)
  - Use special search heuristics
- ▶ Breaking symmetries does not always help (symmetries removed but so might be the left-most solution)

# Bin-Packing Demo ...



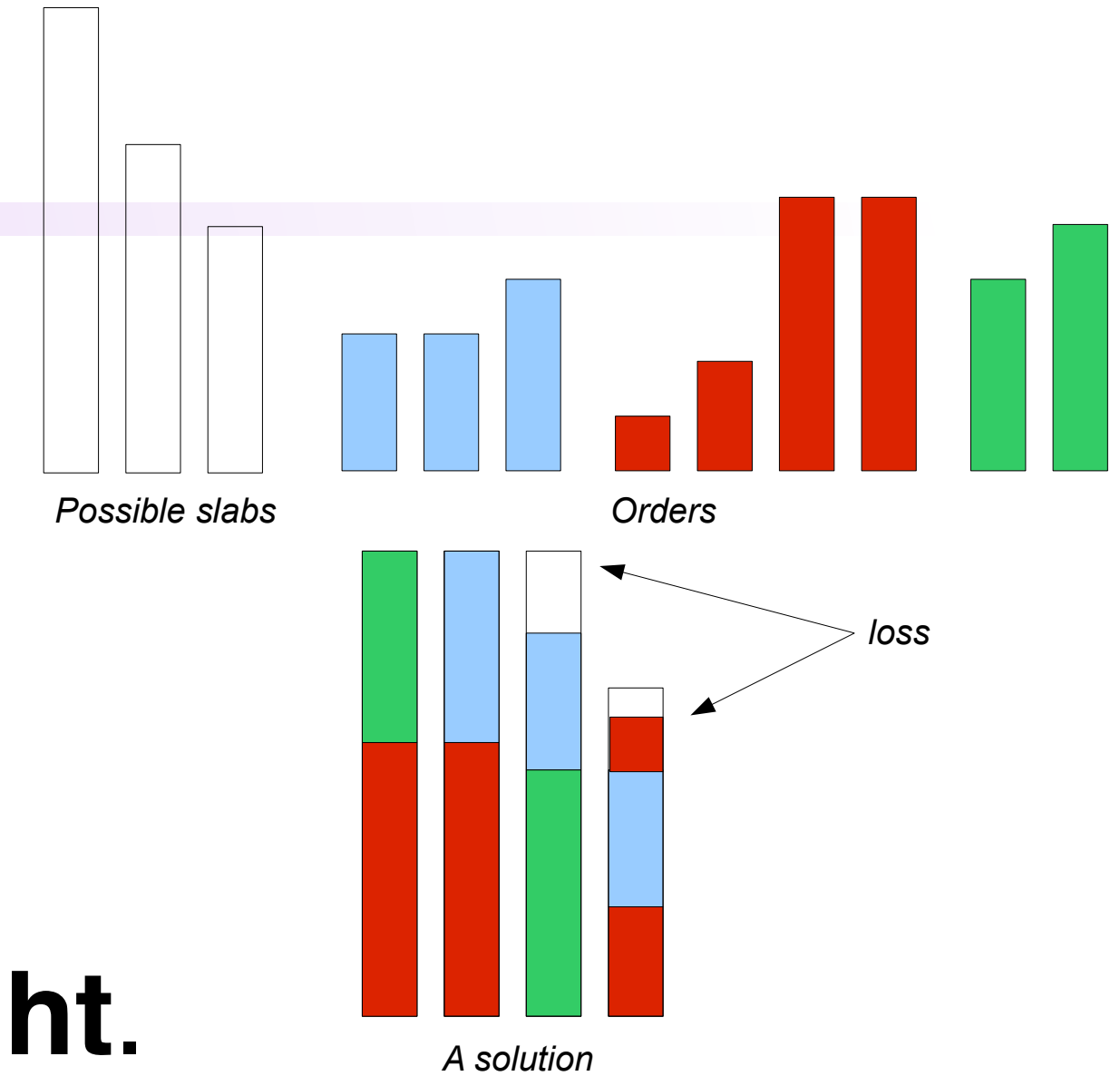
# Steel Mill Slab Problem

(Programming Assignment)

<https://www.csplib.org/Problems/prob038/>

# The Steel Mill Slab Problem

- ▶ Steel produced by casting molten iron into slabs.
- ▶ Only a finite number of slab sizes.
- ▶ An order has two properties,
  - a **color** (route required through the steel mill) and + **weight**.
- ▶ Given n input orders, assign the orders to slabs, the number and size of which are also to be determined, such that **the total weight of steel produced is minimized**.
- ▶ Assignment subject to constraints:
  - Capacity: The total weight assigned to a slab cannot exceed the slab capacity.
  - Colors: Each slab can contain at most 2 colors.



- ▶  $n$  is the number of orders
- ▶  $c_i \in \text{colors}$  is the color of order  $i$
- ▶  $w_i \in \mathbb{N}^+$  is the weight of order  $i$
- ▶  $\sigma$  is the set of different slab capacity. At most  $m$  slabs will be used, we label them from 1 to  $m$  ( $m = n$  if not restricted)

► Decision variables:

- $o_i \in [1..n]$  is the slab attributed to order  $i$

► Auxiliary variables:

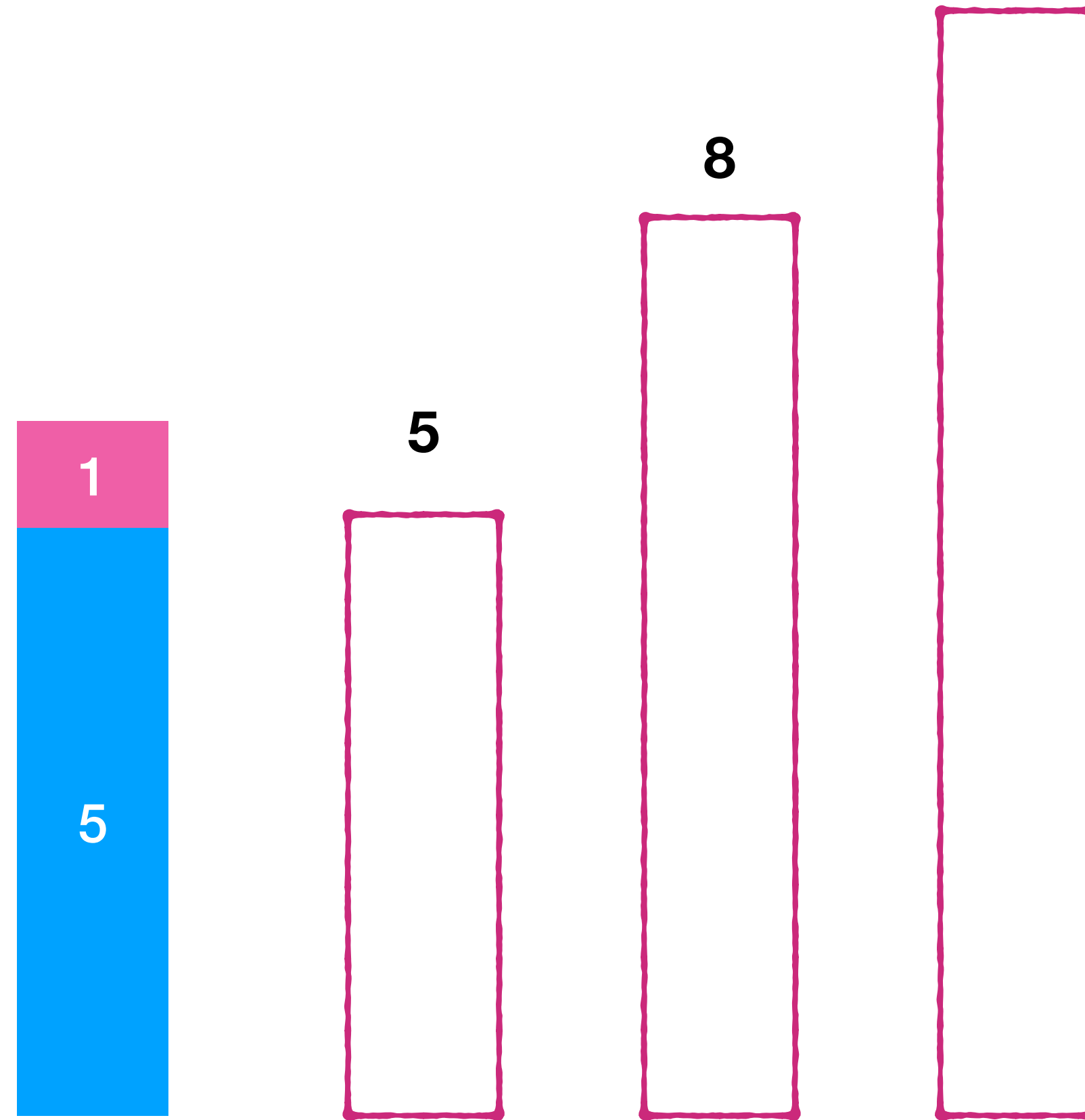
- $p_j \in [0..\text{maxcapa}]$  is the weight of the orders attributed to slab  $j$
- $l_j \in [0..\text{maxcapa}]$  is the minimal loss of slab  $j$  (determined by the slab of minimal capacity  $\geq p_j$ )

► Objective

minimize the total loss:  $\sum_{j \in [1..m]} l_j$

# Computing losses with Element Constraints

- Assume 3 slab capacities  $\{5, 8, 10\}$ , an order of size 5 and 1
- What slab to chose ? What is the loss ?

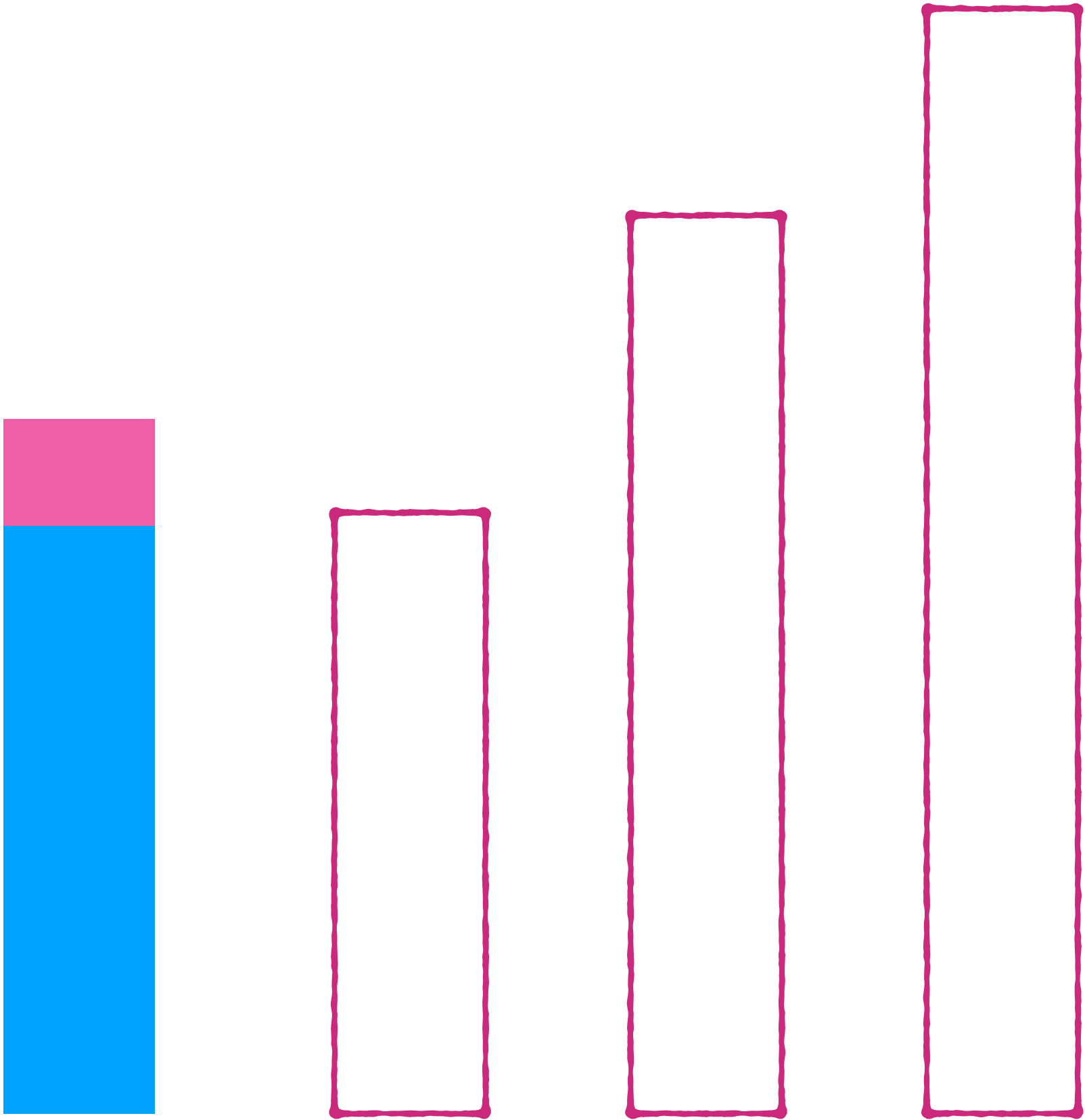


# Computing losses with Element Constraints



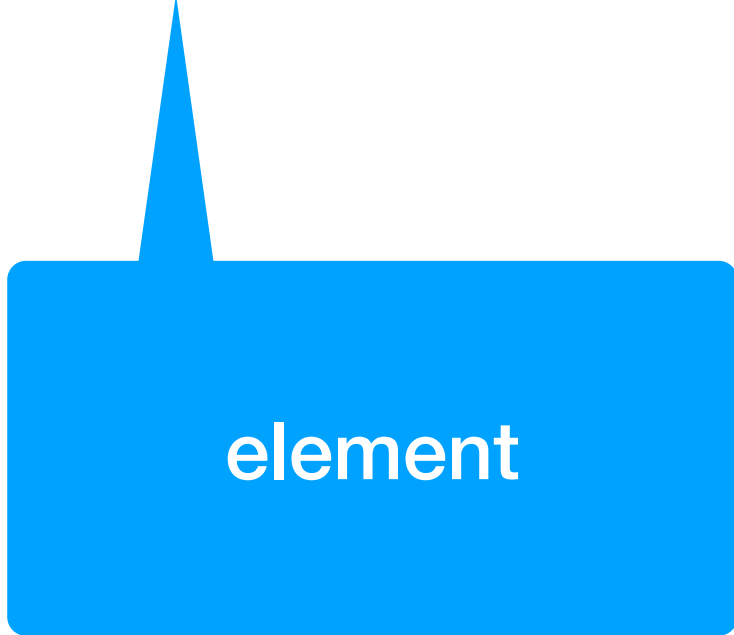
- 💡 precompute the loss for every possible load

| Load | Loss |
|------|------|
| 10   | 0    |
| 9    | 1    |
| 8    | 0    |
| 7    | 1    |
| 6    | 2    |
| 5    | 0    |
| 4    | 1    |
| 3    | 2    |
| 2    | 3    |
| 1    | 4    |
| 0    | 0    |



# Computing losses with Element Constraints

- ▶ Assume 3 slab capacities  $\{5, 8, 10\}$
- ▶ We can preprocess an array  $L = [0, 4, 3, 2, 1, 0, 2, 1, 0, 1, 0]$ .
- ▶ Loss for a total weight  $p_j = 3$ ?  $L = [0, 4, 3, 2, 1, 0, 2, 1, 0, 1, 0]$ .
- ▶ 💡 Use element constraints to link loss and weight variables:  $l_j = L[p_j]$



element

# Computing loads (Bin-Packing or Pack)

$$\forall j \in [1..m] : p_j = \sum_{i \in [1..n]} (o_i = j) \cdot w_i$$



# At most two colors!

Logical Or Constraint (and watched literals)

# Modeling the Color Constraints

## ► At most 2 colors / slab

- Is color  $k$  used in slab  $j$  (yes 1, no 0)?

isOr constraint  
b iff ( $x_1$  or  $x_2$  or ... or  $x_n$ )

$$\bigvee_{i \in [1..n] \mid c_i = k} (o_i = j)$$

isEqual

- $\forall j \in [1..n] :$

$$\sum_{k \in colors} \left( \bigvee_{i \in [1..n] \mid c_i = k} (o_i = j) \right) \leq 2.$$

isOr

# Reified Or: IsOr Constraint

- ▶  $b$  iff  $(x_1 \text{ or } x_2 \text{ or } \dots \text{ or } x_n)$ 
  - $b = \text{true}$ : post  $(x_1 \text{ or } x_2 \text{ or } \dots \text{ or } x_n) = \text{the Or constraint and deactivate}$
  - $b = \text{false}$ : set all variables  $x_i$  to false
  - $x_i$  become true: set  $b$  to true and deactivate (we must listen to all variables)
  - all  $x_i$ 's are false: set  $b$  to false (maintain them in a sparse-set)

# The Or or Clause Constraint

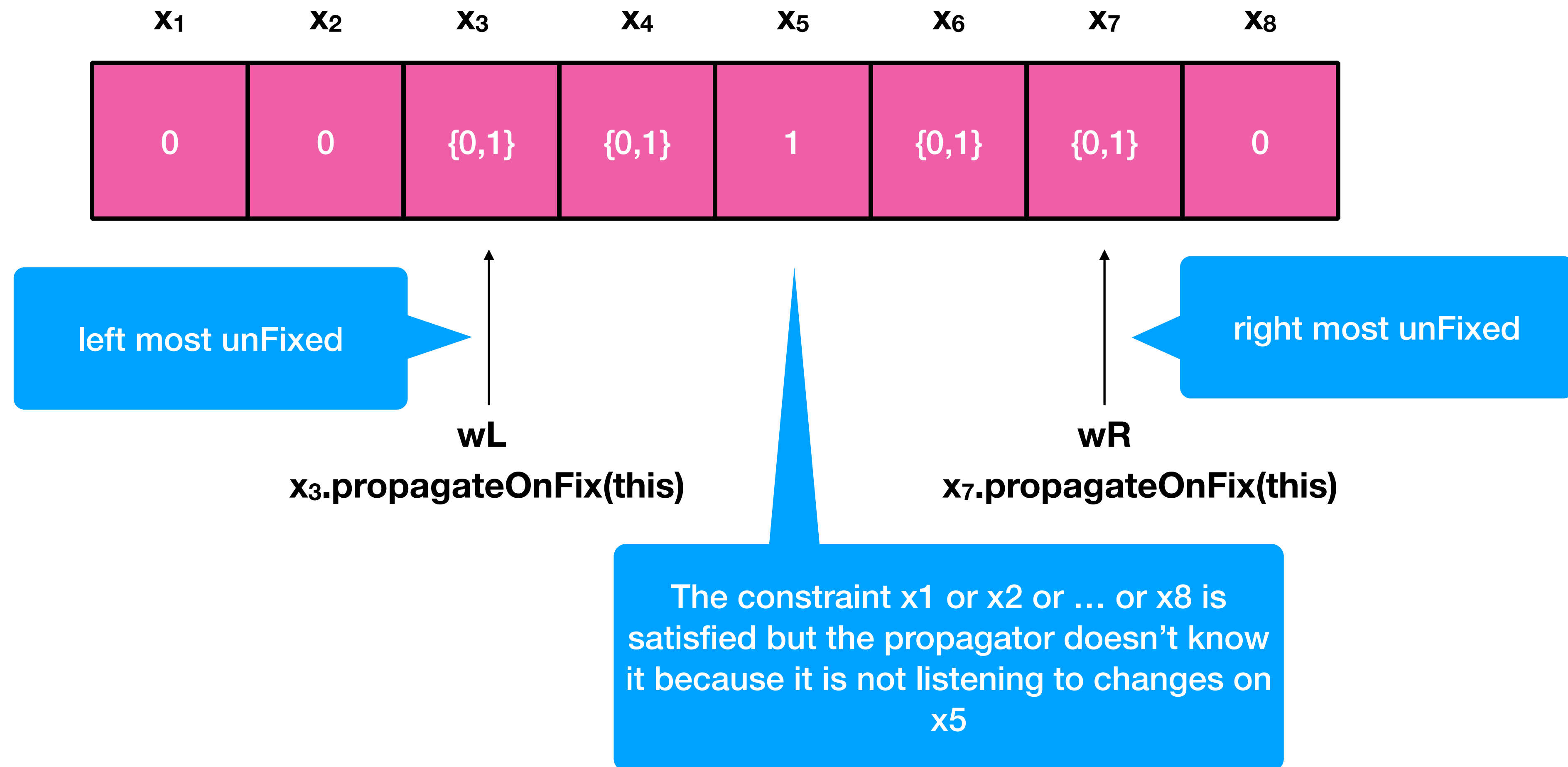
- ▶ At least one boolean variable is true:
  - $x_1$  or  $x_2$  or ... or  $x_n$
- Can only propagate when all variables are false, except one (this is called **unit propagation**).
- This is the only propagation used in modern SAT solvers.

# First implementation

- ▶ Listen to all variables
- ▶ Maintain sparse-set with unbound variables
- ▶ If one variable become true, deactivate the constraint because it is satisfied.
- ▶ If the sparse-set becomes empty and all other variables are false, throw an `InconsistencyException`
- ▶ If only one variable is unbound, the other ones are false, set the last one to true (unit propagation)
- ▶ Can be done with  $O(1)$  per variable change but can we do better?

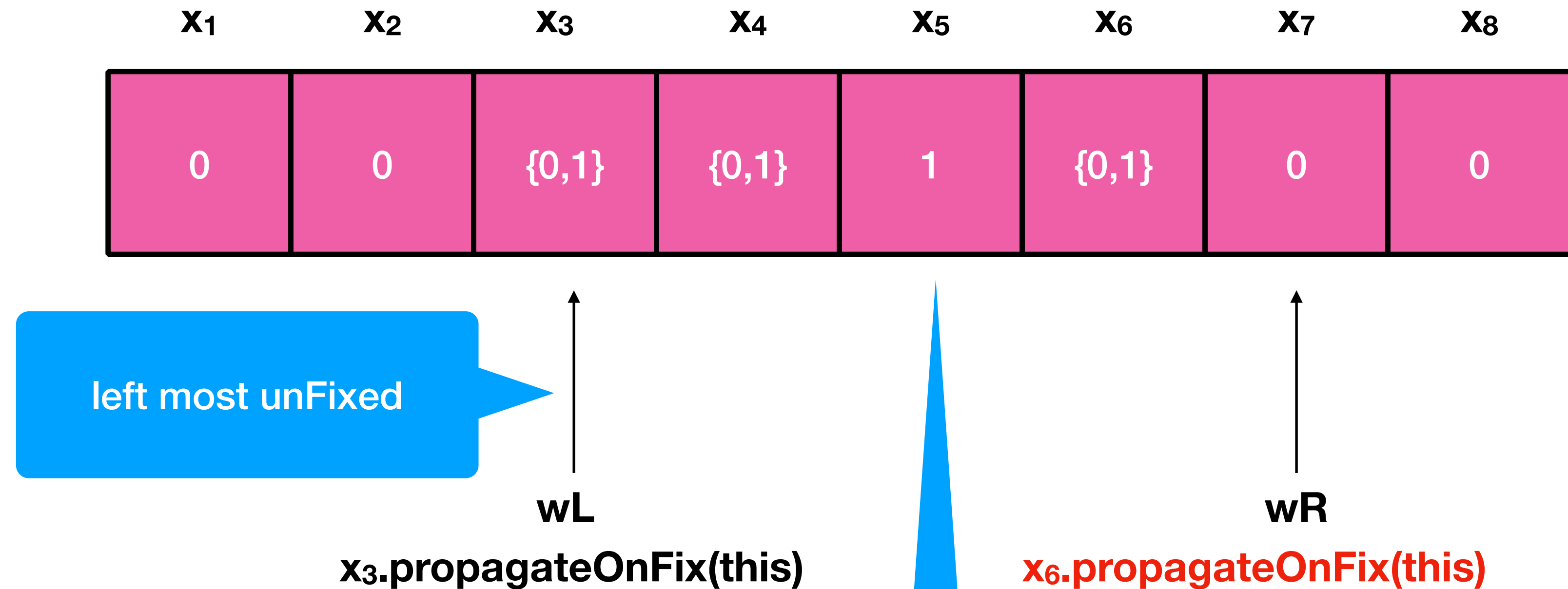
# Watched literal (adapted to MiniCP)

- ▶ Don't listen to all changes, only listening to two variables is enough.
- ▶ Idea: If two variables are either unassigned or assigned true, no need to do anything.



# Watched literal (adapted to MiniCP)

- Don't listen to all changes, only listening to two variables is enough



The constraint x<sub>1</sub> or x<sub>2</sub> or ...  
or x<sub>8</sub> is satisfied but the  
propagator doesn't know it  
because it is not listening to  
changes on x<sub>5</sub>

- ▶ If  $w_L = w_R$  (only one variable  $\neq 0$ ), it must be set to 1
- ▶ If  $w_L > w_R$ , all variable are zero, we must fail (inconsistency).

# Reified Or: IsOr Constraint

- ▶  $b$  iff  $(x_1 \text{ or } x_2 \text{ or } \dots \text{ or } x_n)$ 
  - $b = \text{true}$ : post  $(x_1 \text{ or } x_2 \text{ or } \dots \text{ or } x_n) = \text{the Or constraint and deactivate}$
  - $b = \text{false}$ : set all variables  $x_i$  to false
  - $x_i$  become true: set  $b$  to true and deactivate (we must listen to all variables)
  - all  $x_i$ 's are false: set  $b$  to false (maintain them in a sparse-set)

- ▶ Exercise: bijection on graph coloring for value symmetries
- ▶ Redundant constraints
- ▶ Static vs Dynamic symmetry breaking