Tutorial 3: Tiling with Squares

Let T be the size of the "big" square and $T_1, T_2, \ldots, T_n$ the sizes of n "small" squares. Tile the big square with the small ones such that no two small squares overlap and the big square is totally covered.

1. Write function:

   ```
   solve: int -> int array -> unit
   ```

   that takes the size of the big square and the array of the sizes of the small squares as parameters and solves the tiling problem. This function must:

   - define the domain such that the small squares **remain inside** the big one;
   - constrain the small squares pairwise such that they **don't overlap**, using **reification** operators.

   The number of the instance to solve will be taken as a command line parameter.

2. Display the solution with `gnuplot`: write the coordinates of the four corners of each square in a file, each square being separated by newlines. The following `gnuplot` command reads the file and print the squares:

   ```
   gnuplot> plot "filename" with lines
   ```

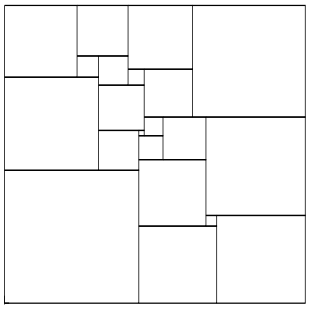

Figure 1: A solution for the third instance

3. Try to solve larger instances by refining your constraint program with:

   - **Redundant constraints** For all horizontal (resp. vertical) lines crossing the small squares $i_1, \ldots, i_p$, we have: $\sum_{j=1}^{p} T_{i_j} = T$.
     Use functional constraint `Interval.is_member x lb ub` which returns a boolean variable set to 1 if `x` belongs to [lb, ub] and 0 otherwise.
   - **More dynamic variable ordering** A standard dynamic variable ordering chooses the next variable to instantiate by minimizing (or maximizing) some criterion, but once the variable is selected, it is not challenged again whenever the assignment fails. The ordering can be "more dynamic" if the criterion is evaluated again after each assignment

(because the removal of a value may refine the domains, so the best variable according to the criterion could change). Predefined goal `Goals.assign` (single non-deterministic assignment) can be used instead of `Goals.indomain` for this purpose.

- **Strategy** Use the "min min" heuristic that first selects the variable with the minimal lower bound.

Compare the performances of the various versions of your solver by printing the **resolution time** and **number of backtracks** on tractable instances.

4. Modify your program such that **all solutions are found** and print their number. Add constraints to **break the permutation symmetry** between small squares of equal size by lexicographically ordering them and compare the number of solutions for the two first instances. **Break the vertical and horizontal geometric symmetry** by constraining the center of the biggest square to lie within the bottom left quarter square, and compare the number of solutions obtained.

5. *Optional question: Break also the symmetry along the second diagonal by constraining that the size of the second square on the x-axis be greater than the one of the second square on the y-axis.*
   Indications: use `FdArray.get` *and* `FdArray.get_cstr` *to obtain the index of the bottom left square, then use its size to obtain the indices of the two second squares and finally order their sizes.*

Data for 4 instances:

T	T_1, T_2, ...
3	2,1,1,1,1,1
19	10,9,7,6,4,4,3,3,3,3,3,2,2,2,1,1,1,1,1,1
112	50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2
175	81,64,56,55,51,43,39,38,35,33,31,30,29,20,18,16,14,9,8,5,4,3,2,1