The objective of this first part of the project is to model a problem for which you will design a custom propagator (in Part 2) and a custom brancher (in Part 3).

For a given \( n > 1 \), the square packing problem consists of finding the smallest enclosing square of size \( s \times s \) such that the \( n \) squares of sizes \( 1 \times 1 \), \( 2 \times 2 \), \ldots, \( n \times n \) are packed without overlapping inside the \( s \times s \) square. (Careful: It thus need not be a perfect packing, as gaps between squares are allowed, hence the problem is very different from Problem 9 of CSPlib (http://csplib.org/), which is solved in the model perfect-square.cpp of the Gecode distribution.)

Consult the following paper by Helmut Simonis and Barry O'Sullivan: Search strategies for rectangle packing, in Peter J. Stuckey (editor), CP 2008, Lecture Notes in Computer Science, volume 5202, pages 52 – 66, Springer-Verlag, 2008. This paper also addresses the more general case of packing the \( n \) squares into a minimal rectangle of size \( w \times h \); to keep this project simple, we here just consider the special case where \( w = s = h \). A preprint of this paper and sample best solutions to the square packing problem can be found at http://4c.ucc.ie/~hsimonis/Search%20Strategies%20for%20Rectangle%20Packing.pdf.

The goal here is to find not only the smallest value for \( s \) but also the position of each square (that is, the \( x \) and \( y \) coordinates of its lower-left corner, the \( x \) axis being horizontal and the \( y \) axis vertical) in the enclosing \( s \times s \) square, whose origin is at coordinates \((0, 0)\). The model described in the paper uses a DISJOINT2 constraint for enforcing that the squares do not overlap, and the CUMULATIVE constraint as an implied constraint. We will here use a more naive model that expresses both constraints by a collection of reified constraints. Perform the following tasks:

- Create an IntVar called \( s \) to denote the size of the enclosing square and two IntVarArray of integer variables to denote the \( x \) and \( y \) coordinates of the lower-left corner of each enclosed square.
  
  **Hint:** Think carefully how large a coordinate for a square can be. A good idea is to define a static member function size that returns the size of the square with a given number. Since it will be helpful for branching, index the squares so that the square with number \( 0 \) has size \( n \), the square with number \( 1 \) has size \( n - 1 \), and so on.

- Express with reification that no two squares overlap. Two squares \( s_1 \) and \( s_2 \) do not overlap if and only if \( s_1 \) is left of \( s_2 \), or \( s_2 \) is left of \( s_1 \), or \( s_1 \) is above \( s_2 \), or \( s_2 \) is above \( s_1 \). To express these geometrical relations, use reified constraints taking the coordinates and sizes of the squares into account.
• Read the paper and think carefully which additional constraints you need to post (when \( w = s = h \)). At least incorporate the following ideas: problem decomposition (Section 2.1), symmetry removal (Section 2.2), empty strip dominance (only initial domain reduction, Section 4.1), and ignoring \( 1 \times 1 \) squares (Section 4.2).

**Hint:** The sum of the first \( n \) square sizes is: \( \sum_{i=1}^{n} i^2 = \frac{n \cdot (n + 1) \cdot (2 \cdot n + 1)}{6} \).

• Consider the column with coordinate \( x \): The sum of the sizes of the squares occupying space at column \( x \) must be less than or equal to \( s \). Express this implied constraint for all columns by using reified constraints. Do the same for all rows. As \( s \) is a decision variable, you have to consider all columns and rows from 0 to \( s \text{max}() \).

**Hint:** The reified \( \text{Dom} \) constraint might come in handy here.

• Design a good branching heuristic. Aspects you want to take into account are: Always branch on \( s \) first. Try to assign first all \( x \) coordinates, and then all \( y \) coordinates. Try bigger squares first. Try to place squares by increasing (or decreasing) coordinates. Express your branching heuristic with predefined variable and value selectors; in Part 3 of this project, you will design a custom brancher that follows some of the ideas described in Section 3.6. (Note that you are only asked for one branching heuristic.)

• Give the runtimes and numbers of failures until the first optimal solution for a reasonable interval of values for \( n \). If randomisation is involved in a heuristic, then report the average statistics over at least 10 independent runs. All runtimes are to be given in seconds, with the same reasonable precision. Use a reasonable time-out value. All statistics can be given in tabular or graphic form.

Your program must be called `squarePacking1.cpp` and take a number \( n \) of squares as command-line argument. For team \( t \), the report must be called `project1-team\( t \).pdf` and the command-line call `squarePacking1 \( n \)` must produce a textfile called `project-team\( t \)-\( n \).txt` with the first solution, which has the structure of the sample file `http://www.it.uu.se/edu/course/homepage/consprog/ht14/homeworks/project-team9-n17.txt` (but not necessarily the same contents for \( n = 17 \)), the runtime being expressed in seconds. You may want to use the Python script (written by former student Shervin Shoravi) at `http://www.it.uu.se/edu/course/homepage/consprog/ht13/resources/project` in order to visualise solutions.

**Submission Instructions**

All program documentation and question answers (other than the programs) **must** be in a single report in PDF format; all other formats are rejected.

• Take Section 1.1 of the demo report at `http://www.it.uu.se/edu/course/homepage/consprog/ht14/resources/demoReport` as a strict guideline for the structure and content of a model description in the report, as well as an indication of its expected quality of content.

• All models must be described in pseudo-code and plain English, as in the demo report.

• State the question and task number (if any) of each answer in the report.

• Write clear answers, programs, and documentation.

• Justify all claims and answers, except where explicitly not required.
• State any assumptions you make.

• Thoroughly proof-read, spell-check, and grammar-check your report.

• Upload all code that is required to compile and run your programs.

• Match exactly the uppercase, lowercase, layout, and spacing conventions of any file names and structures imposed by the questions, as we reserve the right to process your program automatically: no grade of 4 or 5 will be awarded if manual intervention is necessary.

Only one of the teammates submits the files (report and programs), without folder structure and without compression, via the Student Portal (whose clock may differ from yours) by the given hard deadline.

Grading

If all questions have been seriously attempted and all requested programs exist in files with the imposed names, have the documentation prescribed in the demo report, compile without error under gcc, and produce correct outputs to some of our grading tests in reasonable time under version 4.2.1 of Gecode on one of the computers of the IT department, then you get an initial score of at least 1 point:

• If your programs pass most of our grading tests, then you get an initial score of 4 or 5 points, depending also on the quality of the report; you are not invited to the grading session and your initial score is your final score.

• If your programs fail many of our grading tests, then you get an initial score of 1 or 2 points, depending also on the quality of the report; you are invited to the grading session, where you can increase your initial score by 1 point into your final score.

Otherwise you get a final score of 0 points and fail this part of the course.