Elementary MiniZinc for Mathematics and Puzzles

Charles Joscelyne

November 28, 2020

Preface

MiniZinc: *https://www.MiniZinc.org/* is a free and open source constraint solving programming language. Concisely put, MiniZinc allows a user to model a quantitative problem as a collection of constraints expressed in terms of modeler defined decision variables. Once the constraints have been defined, MiniZinc calls on its library of pre-programmed solvers to find one or more solutions of the variables satisfying the constraints. Being more akin to a functional programming language than an imperative one, MiniZinc transfers the burden of writing a specific combinatorial or optimization algorithm imperatively to that of expressing a given problem functionally in terms of MiniZinc language constructs.

By way of example let us consider problem 13 from the 2019 American Mathematics Competition 8 Examination.

```
A palindrome is a number that has the same value when read from
left to right or from right to left. (For example, 12321 is a
palindrome.) Let N be the least three-digit integer which is not a
palindrome, but which is the sum of three distinct two-digit
palindromes. What is the sum of the digits of N?

%Decision variables
var 1..9:N1;
var 0..9:N2;
var 0..9:N3;

var 1..9:A;
var 1..9:B;
var 1..9:C;


%Constraints
constraint N1 != N3;
constraint A != B;
constraint A != C;
constraint B != C;
constraint 100*N1 + 10*N2 + N3 = 10*A+A+10*B+B+10*C+C;



%solve
solve minimize 100*N1 + 10*N2 + N3;

%Display solution
output [ "\(N1)"++" "++"\(N2)"++" "++"\(N3)"++" sum of digits = "+.
```

```
+"\(N1+N2+N3)"];
% 1 1 0 sum of digits = 2
```

The solution: 110 = 77 + 22 + 11 to the palindrome problem is referenced as a MiniZinc comment in the program's last line.

A cursory glance of the program above shows the three components of any elementary MiniZinc model:

1. Definition of the model's decision variable(s).

2. Definition of model's decision constraints.

3. A solve statement.

Although an output statement is included in above program, MiniZinc will usually display useful results in the absence of one. Nonetheless, a MiniZinc modeler most likely will want to control the contents and format of any output. MiniZinc's output statement enables a modeler to do so.

Before proceeding to the study of individual problems and their solution in MiniZinc, I should note that this tract is not entirely intended to be a tutorial on MiniZinc. Although, I expect the reader would be able to learn many basic features of the MiniZinc language here, the coverage of the language is far from exhaustive. For those new to MiniZinc, Coursera offers an introductory course that can be found here: https://www.coursera.org/learn/basic-modeling. A somewhat concise, but nonetheless useful tutorial can be found here: https://www.MiniZinc.org/tutorial/MiniZinc-tute.pdf. My aim here rather is to solve what I view to be interesting problems, employing where possible, only the more elementary constructs of MiniZinc. A challenge one faces when learning and subsequently using any programming language is to have an ample supply of accessible code samples upon which to study and apply. I believe code samples are even more vital when programming in a functional styled language like MiniZinc. Whereas no more than few examples of

loop constructs or conditional branch statements might suffice when learning to program in Java or Python, the same is not true when first approaching a functional or functional styled language like Haskell, R, or MiniZinc. Learning to program functionally is best supported through accessible and at the same time interesting code samples. Unfortunately, it has been my experience that many sample MiniZinc models, due to the complexity of the problem being addressed, tend to obscure the language features employed. This makes it less likely for one new to the language to be successful later in applying what has been presented in a novel setting. Hopefully, the nature of the problems solved here in MiniZinc will serve as a compendium of readily accessible examples, which one can refer to in order to adapt or extend.

In the following pages, problems are categorized by the language features employed in the problem's solution. This is opposed to the common grouping by problem content. In this way, a MiniZinc modeler when needing a particular language feature will be able to more easily index one or more relevant solved examples. This approach, of course, is the one familiarly taken by many programming language expositions along with dictionaries and lexicons where sample sentences illustrate a term's usage. The contents section given below can then be seen to be organized in this manner.

A constraint solver like MiniZinc is a powerful tool. As one becomes more proficient in its use, one's appreciation of scope and range of problems amenable to being solved with the language will surely grow. In fact, one could possibly be lulled into adopting the (unrealistic) mindset that every combinatorial or optimization problem encountered can be viewed as a constraint problem solvable by MiniZinc. The possibility of acquiring such a narrow view should not raise alarm for the liberally educated who will surely consign MiniZinc to be just another problem solving tool in their kit.

When time and energy permits, I encourage the reader to attempt to solve a problem before referencing the MiniZinc solution. One should realize that for almost all the problems presented here, a purely mathematical solution exists, which for the so inclined can be of equal interest to discover. To help the reader to gain more experience using MiniZinc, a collection of companion exercises and their solutions are presented at this tract's end.

Before proceeding to the problems and their solution in MiniZinc, I will point out that all the MiniZinc code presented here is linked to authors GitHub account, which can be found here: https://github.com/pjoscely.

Contents

Modeling with Integer Variables

For our initial example, let us consider the following high school algebra coin problem:

A bag contains twice as many pennies as nickels and four more dimes than quarters. Find all possibilities for the number of each coin if their total value is $2.01.

```
%Coin variables
var 0..201:P;
var 0..41:N;
var 0..21:D;
var 0..8:Q;

%twice as many pennies as nickels
constraint P = 2*N;

%four more dimes than quarters
constraint D = Q + 4;

%total value is $2.01
constraint 201 = P + 5*N + 10*D + 25*Q;

%required solve statement
solve satisfy;

%display results
output["P = "++"\(P)"++" N = "++"\(N)"++" D = "++"\(D)"++" Q = "++"\.
(Q)"];

/*
P = 46 N = 23 D = 4 Q = 0
----------
P = 36 N = 18 D = 5 Q = 1
----------
P = 26 N = 13 D = 6 Q = 2
----------
P = 16 N = 8 D = 7 Q = 3
----------
P = 6 N = 3 D = 8 Q = 4
----------
==========
*/
https://github.com/pjoscely/Math-Comp-MiniZinc/blob/master/AMC%208/
Alg%202%20H%20Coin%20Problem.mzn
```

The program's output/solution is included as a comment after the code. Single line comments in MiniZinc begin with a "%", while multiline comments begin with "/*" and end with "*/". The double dashed line concluding the output's indicates

that MiniZinc found no more solutions. While the model is self-explanatory, it should be pointed out that the defined decision variables are declared with integer bounds in accordance with the problem. Doing so, narrows the search and lessens the final number of constraint statements. More on this later.

The following is problem 7 from the 2019 American Mathematics Competition 8 Examination:

```
Shauna takes five tests, each worth a maximum of 100 points.
Her scores on the first three tests are 76, 94, and 87.
In order to average 81 for all five tests, what is the lowest score
she could earn on one of the other two tests?

(A)  48  (B)  52  (C)  66  (D) 70  (E) 74


%test scores
par int: first = 76;
par int: second = 94;
par int: third = 87;

var int: fourth;
var int: fifth;

%bounds on possible test scores
constraint fourth >= 0;
constraint fifth >= 0;
constraint fourth < 101;
constraint fifth < 101;

%final average constraint
constraint (first + second + third + fourth + fifth)/5 = 81;

%minimize fourth test score
solve minimize fourth;

%display result
output [" lowest fourth score = "++"\(fourth)"]

 %lowest score = 48

https://github.com/pjoscely/Math-Comp-MiniZinc/blob/master/
AMC%208/amc8_2019_7.mzn
```

This model introduces the shorthand "par" for a fixed parameter declaration. The decision variables fourth and fifth are declared as "int", and are constrained later in the model. Like the model presented in the preface, this model introduces

the very useful "solve minimize" command, which has its counterpart "solve maximize".

A small-sized knapsack problem is problem 2 from the 2018 American Mathematics Competition 12A Examination:

While exploring a cave, Carl comes across a collection of 5-pound rocks worth 14 each, 4-pound rocks worth 11 each, and 1-pound rocks worth 2 each. There are at least 20 of each size. He can carry at most 18 pounds. What is the maximum value, in dollars, of the rocks he can carry out of the cave?

(A) 48   (B) 49   (C) 50   (D) 51   (E) 52

```
%decision variables
var int:five_p;
var int:four_p;
var int:one_p;

%positive constraints
constraint five_p >=0;
constraint four_p >=0;
constraint one_p >=0;

%define a wt variable
var int:wt = 5*five_p+4*four_p+one_p;

%carry at most 18 pounds
constraint wt<=18;

%maximum value, in dollars
solve maximize 14*five_p+11*four_p+2*one_p;

%display the result
output["five_p = "++"\(five_p)"++" four_p = "++"\(four_p)"++" one_p = "+
+"\.
(one_p)"++ " Max value = "++"\(wt)"];


% five_p = 2 four_p = 2 one_p = 0 Max value = 18
```

https://github.com/pjoscely/Math-Comp-MiniZinc/blob/master/
AMC%2012A/amc12A_2018_2.mzn

MiniZinc found the maximum value of 18 that is obtained by placing two 5-pound rocks and two 4-pound rocks in the knapsack. Although it is customary to place constraints after variable definitions, this is not required by MiniZinc as can

be seen by the location in the code of the definition of the "wt" variable. Lastly, the three positive constraints insures the model will terminate.

Problem 21 from the 2018 American Mathematics Competition 8 Examination demonstrates the use of MiniZinc's "mod" function:

```
How many positive three-digit integers have a remainder of 2
when divided by 6, a remainder of 5 when divided by 9,
and a remainder of 7 when divided by 11?

(A)  1   (B)   2   (C)   3   (D)   4   (E)   5

%possible positive three-digit integers
var 100..999:n;

%remainder constraints
constraint n mod 6 = 2;
constraint n mod 9 = 5;
constraint n mod 11 = 7;

solve satisfy;

%display result
output ["n = "++"\(n)"]

n = 194;
n = 392;
n = 590;
n = 788;
n = 986;

https://github.com/pjoscely/Math-Comp-MiniZinc/blob/master/AMC%208/
amc8_2018_21.mzn
```

Modeling with Float Variables

MiniZinc has the capability of handling "real number" constraint solving using floating point solving. For the simple LP problem below, one needs to change the default configuration. If using the MiniZinc IDE this is accomplished by opening the Configuration tab and under solving, one then selects G12 MIP:

```
Find the maximal value of z = 3x + 4y
subject to the following constraints:

x + 2y <= 14, 3x - y >= 0, x - y <= 2

%define float variables
var float:x;
var float:y;

%constraints
constraint x+2*y<=14;
constraint 3*x-y>=0;
constraint x-y<=2;

%maximize
solve maximize 3*x+4*y;

%display the result
output["x = "++"\(x)"++" y = "++"\(y)"++" Max = "++"\(3*x+4*y)"];

%x = 6.0 y = 4.0 Max = 34.0
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/LP1.mzn

The maximal solution is supplied as a comment on the program's last line.

Matt Parker, the "Stand-Up Mathematician", features a bi-monthly puzzle on his Thinks Maths website. His puzzle #11: https://www.think-maths.co.uk/agepuzzle is given below. A MiniZinc model solution, which employs floating point variables follows the puzzle statement. Again, should the reader wish to run this model the G12 MIP option should be selected upon opening the Configuration tab:

David and Anton's ages combined equals 65. David is currently three times as old as Anton was when David was half as old as Anton will be when Anton is three times as old as David was when David was three times as old as Anton.

Puzzle for submission: How old is David?

```
% David's ages
var float :D0;
var float :D1;
var float :D2;
var float :D3;

% Anton's ages
var float :A0;
var float :A1;
var float :A2;
var float :A3;

% ages combined equals 65
constraint D0+A0 = 65;

% David is currently three times as old as Anton was when
constraint D0 = 3*A1;

% David was half as old as Anton will be when
constraint D1 = (0.5)*A2;

% Anton is three times as old as David was when
constraint A2 = 3*D3;

% David was three times as old as Anton
constraint D3 = 3*A3;

% constant difference in ages
% across time instances
constraint D1-D0 = A1-A0;
constraint D2-D1 = A2-A1;
constraint D3-D2 = A3-A2;

solve satisfy;

output
[
  "David age's: " ++ show(D0)
];

%David age's: 37.5
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/Matt_P_Age.mzn

As is the case for many problems, once the appropriate variables have been defined, a problem statement has a seemingly direct translation into MiniZinc constraints. David's age is supplied as a comment on the program's last line.

Consider now the following Fractional Knapsack problem. In this case, items can be broken into smaller pieces, meaning that we can select fractions of items.

Let us suppose that the capacity of the knapsack is $W = 60$ and the list of provided items are shown in the following table: https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_fractional_knapsack.ht

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24

Like the two floating point models above, this model requires us to select under solving the G12 MIP option:

```
%define float variables
var 0.0..1.0:A;
var 0.0..1.0:B;
var 0.0..1.0:C;
var 0.0..1.0:D;

%sack can contain no more than 60
constraint 40*A+10*B+20*C+24*D<=60;

%maximize profit
solve maximize 280*A+100*B+120*C+120*D;

%display the result
output["A = "++show_float(1, 2, A)++" B = "++show_float(1, 2, B)++
       " C = "++show_float(1, 2, C)++" D = "++show_float(1, 2, D)++
       " Max = "++show_float(5, 2,280*A+100*B+120*C+120*D)];

%A = 1.00 B = 1.00 C = 0.50 D = 0.00 Max = 440.00
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/frac_knapsack.mzn

MiniZinc returns the optimal solution of one unit of A and B and a half unit of C. D is not taken. The Fractional Knapsack problem may be solved with a greedy algorithm in polynomial time whereas the classic knapsack problem is NP-hard. These issues, although vitally important for the imperative programmer working in

say Java, are not transparent when using MiniZinc. As MiniZinc relies on standard techniques like backtracking and constraint programming the user should keep in mind that MiniZinc is not a panacea. Like any other specially crafted imperative program, in the absence of an efficient algorithm, a MiniZinc program's performance will suffer when negotiating very large search spaces.

The formula below is used to calculate the fixed monthly payment (P) required to fully amortize a loan of L dollars over a term of n months at a monthly interest rate of c. [If the quoted rate is 6%, for example, c is .06/12 or .005].

$$P = \frac{L[c(1 + c)^n]}{[(1 + c)^n - 1]}$$

The Minizinc code given below calculates the allowable home loan given a set monthly payment, interest rate, and time period. Since the power function "pow" apparently does not allow for decision variables, the formula is used here only to find the allowable loan. It should be pointed out that MiniZinc allows for the easy creation of a separate data file containing given parameter values. Details on how to accomplish this can be found here: https://www.MiniZinc.org/tutorial/MiniZinc-tute.pdf.

```
%Monthly payments
par float:P = 1200.0;

%Interest rate divided over 12
par float:c = 0.002417; % 2.9 percent divided by 12

%30 years times twelve months
par float:n = 360.0;

%Loan allowed
var float:L;
%Mortgage equation

constraint P = L*(c*pow(1+c,n))/(pow(1+c,n)-1);

solve satisfy;
```

```
%display the result
output["Maximum loan = $"++show_float(6, 2, L)];

Maximum loan = $288,287.32
```

A maximum loan of $288,287.32 can be had for monthly payments of $1200, amortized monthly over 30 years, at an interest rate of 2.9%.

Nonlinear optimization problems can sometimes be solved naively with MiniZinc, provided initial bounds are put on the decision variables. Below is a problem that is usually handled with Lagrange Multipliers. Interestingly, this model fails under the G12 MIP option, but yields a solution if the default Geocode(bundled) option is selected:

Maximize $81x^2 + y^2$ subject to the constraint $4x^2 + y^2 = 9$

```
%define float variables
var -1.5..1.5:x;
var -3.0..3.0:y;

constraint 4*x*x+y*y = 9;

solve maximize 81*x*x+y*y;


%display the result
output["x = "++show_float(6, 1, x)++" y = "++show_float(6, 2, y)++
      " Max = "++show_float(6, 2, 81*x*x+y*y)];

%x =    -1.5 y =   -0.00 Max = 182.25
```

When using MiniZinc, it is a good practice to bound decision variables whenever possible. For this model, bounding the variables is actually required for the model to terminate. Since all variables are squared, the additional solution x = 1.5 y = 0 is easily seen by inspection.

Although, it is not entirely our purpose here to categorize problems, which can or cannot be solved with MiniZinc, it is interestingly to note that a similar problem given below fails to return a solution in any reasonable time.

Minimize $x^2 + 2y^2 - 4y$ subject to the constraint $x^2 + y^2 = 9$

```
%define float variables
var -3.0..3.0:x;
var -3.0..3.0:y;

constraint x*x+y*y = 9;

solve m x*x +2*y*y-4*y;

%display the result
output["x = "++show_float(6, 1, x)++" y = "++show_float(6, 2, y)++
       " Max = "++show_float(6, 2, x*x +2*y*y-4*y)];
```

Modeling with Conditionals

Like other programming languages MiniZinc supports the formation of Boolean statements. These are indispensable when tailoring constraints for a particular model. The usual "and", "or", and "not" from say Python are expressed in MiniZinc as "/\", "\/", and "!" respectively.

For our first example, we employ the Boolean not "!" operator in MiniZinc, to solve a simple map coloring problem. Here we color the 6 New England States with three colors, the colors are the enumerated date types: "red", "green", "blue". This example is adapted from: https://www.minizinc.org/doc-2.4.3/en/modelling.html#sec-modelling

```
% Coloring New England using three colors
enum c = {red, green, blue};

% State variables
var c: Maine;
var c: NewHamp;
var c: Vermont;
var c: Conn;
var c: Mass;
var c: RhodeI;

% bordering states have different colors
constraint Maine != NewHamp;
constraint NewHamp != Vermont;
constraint NewHamp != Mass;
constraint Vermont != Mass;
constraint Mass != Conn;
constraint Mass !=  RhodeI;
constraint Conn != RhodeI;

solve satisfy;

Maine = blue;
NewHamp = green;
Vermont = red;
Conn = red;
Mass = blue;
RhodeI = green;

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
new_england_map.mzn
```

Opening the Configuration tab and changing the under user defined behavior to display all solutions, MiniZinc displays 24 distinct three color colorings of which one is given above. If only two colors are used then MiniZinc reports that the problem is unsatisfiable.

For the next example, consider the July 2018 IBM Ponder This Challenge:

Let's call a triplet of natural numbers "obscure" if one cannot uniquely deduce them from their sum and product. For example, {2,8,9} is an obscure triplet, because {3,4,12} shares the same sum (19) and the same product (144). Find a triplet of ages {a, b, c} that is obscure and stays obscure for three more years: {a+1,b+1,c+1}, {a+2,b+2,c+2} and {a+3,b+3,c+3}.
https://www.research.ibm.com/haifa/ponderthis/challenges/July2018.html

```
% Decision variables
var 1..120: a;
var 1..120: b;
var 1..120: c;

var 0..120: x1;
var 0..120: x2;
var 0..120: x3;

var 0..120: y1;
var 0..120: y2;
var 0..120: y3;

var 0..120: z1;
var 0..120: z2;
var 0..120: z3;

var 0..120: w1;
var 0..120: w2;
var 0..120: w3;

% Required constraints
constraint a+b+c = x1+x2+x3;
constraint a*b*c = x1*x2*x3;
constraint a != x1 \/ b != x2 \/c != x3;
constraint a <= b /\ b <= c;
constraint x1 <= x2 /\ x2 <= x3;

constraint a+b+c+3 = y1+y2+y3;
constraint (a+1)*(b+1)*(c+1) = (y1)*(y2)*(y3);
constraint a+1 != y1 \/ b+1 != y2 \/ c+1 != y3;
constraint y1 <= y2 /\ y2 <= y3;
```

```
constraint a+b+c+6 = z1+z2+z3;
constraint (a+2)*(b+2)*(c+2) = (z1)*(z2)*(z3);
constraint a+2 != z1 \/ b+2 != z2 \/ c+2 != z3;
constraint z1 <= z2 /\ z2 <= z3;

constraint a+b+c+9 = w1+w2+w3;
constraint (a+3)*(b+3)*(c+3) = (w1)*(w2)*(w3);
constraint a+3 != w1 \/ b+3 != w2 \/ c+3 != w3;
constraint w1 <= w2 /\ w2 <= w3;

solve satisfy;
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
JulyIBM.mzn

One of the many solutions found by this MiniZinc model is given in the table below:

obscure triplet	sum	product	shared triplet
{15, 22, 48}	85	15840	{12, 33, 40}
{16, 23, 49}	88	18032	{14, 28, 46}
{17, 24, 50}	91	20400	{20, 20, 51}
{18, 25, 51}	94	22950	{15, 34, 45}

This IBM puzzle perhaps has some intrinsic interest, but one would imagine that a reader relatively new to MiniZinc might be more interested in the use of the language's boolean operators in composing constraints. The model's constraint code is glaringly redundant, and the copious use of similarly named variables makes one wonder if either arrays or functions exist in MiniZinc. While the question of functions will be taken up in a subsequent work, arrays will be discussed in the next section. Nonetheless, in the spirit of the this tract, the code is presented as is. Should the reader wish to run this model the default Geocode(bundled) option should be selected.

Along with Boolean statements, MiniZinc allows for conditional branch statements like those found in imperative programming languages. For an

example, let us consider problem 18 from the 2018 American Mathematics

Competition 8 Examination: https://artofproblemsolving.com/wiki/index.php/

2018_AMC_8_Problems/Problem_19#Problem_19

In a sign pyramid a cell gets a "+" if the two cells below it have the same sign, and it gets a "-" if the two cells below it have different signs. The diagram below illustrates a sign pyramid with four levels. How many possible ways are there to fill the four cells in the bottom row to produce a "+" at the top of the pyramid?

```
                    +
                +       +
            -       -       -
        +       -       +       -
```
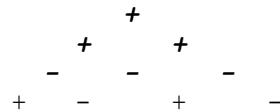
(A)   2   (B)   4   (C)   8   (D)   12   (E)   16

```
% Model + by 0 - by 1
var 0..1:row11;

var 0..1:row21;
var 0..1:row22;

var 0..1:row31;
var 0..1:row32;
var 0..1:row33;

var 0..1:row41;
var 0..1:row42;
var 0..1:row43;
var 0..1:row44;

% Initialize the first row
constraint row11 = 0;

% Pyramid cell constraints
constraint if row21 = row22 then row11 = 0 else row11 = 1 endif;
constraint if row31 = row32 then row21 = 0 else row21 = 1 endif;
constraint if row32 = row33 then row22 = 0 else row22 = 1 endif;
constraint if row41 = row42 then row31 = 0 else row31 = 1 endif;
constraint if row42 = row43 then row32 = 0 else row32 = 1 endif;
constraint if row43 = row44 then row33 = 0 else row33 = 1 endif;
solve satisfy;

output["\(row41)"++" "++"\(row42)"++" "++"\(row43)"++" "++"\(row44)"];

0 0 0 0
1 0 0 1
1 1 0 0
0 1 0 1
1 0 1 0
0 0 1 1
0 1 1 0
1 1 1 1
```

As can be seen from above the model finds a total of 8 ways to fill the bottom row and satisfy the problem's requirements. Here the plus (+) sign is represented by a 0 and the minus (-) sign is represented by a 1 in the model. The syntax of MiniZinc's "if then endif" construct can be gleaned from its use above. Should the reader need more details, on "if", nested "if" statements, and other conditional statements in MiniZinc, they can be found here: https://www.minizinc.org/doc-2.4.3/en/modelling2.html#conditional-expressions.

Modeling with Arrays

As was seen, for some of the the models developed above the number of variables employed became quite large. The more complex the problem one is faced with, the more likely it is that one will require more than a modicum of variables and consequent constraints to achieve a solution. As with imperative programming languages, MiniZinc allows for the creation of data structures such as arrays and sets and their incorporation in constraints to help manage these situations. In fact, as is noted in the MiniZinc tutorial, one is often "interested in building models where the number of constraints and variables is dependent on the input data". Arrays and sets will also allow us this flexibility to model variable size inputs.

For those already accustomed to arrays in imperative languages, MiniZinc presents little in the way that is new or different. Suffice it to say that arrays in MiniZinc mirror those found in imperative languages, excepting that array indices start with one instead of zero. For our next MiniZinc array example, let us consider problem 25 from the 2017 American Mathematics Competition 10B Examination:

Last year Isabella took 7 math tests and received 7 different scores, each an integer between 91 and 100, inclusive. After each test she noticed that the average of her test scores was an integer. Her score on the seventh test was 95. What was her score on the sixth test?

(A)   92   (B)   94   (C)   96   (D)   98   (E)   100

```
include "alldifferent.mzn";

% array for possible test scores
array[1..7] of var 91..100: d;

%After each test the average of the test scores is an integer.
constraint (d[1]+d[2]) mod 2 = 0;
constraint (d[1]+d[2]+d[3]) mod 3 = 0;
constraint (d[1]+d[2]+d[3]+d[4]) mod 4 = 0;
constraint (d[1]+d[2]+d[3]+d[4]+d[5]) mod 5 = 0;
constraint (d[1]+d[2]+d[3]+d[4]+d[5]+d[6]) mod 6 = 0;
constraint (d[1]+d[2]+d[3]+d[4]+d[5]+d[6]+d[7]) mod 7 = 0;

%Her score on the seventh test was 95
constraint d[7] =  95;
```

```
%7 different scores in all
constraint alldifferent(d);

solve satisfy;

% display scores
output["6th score: "++"\(d[6])"++" "++" Scores: "++"\(d)"];

6th score: 100  Scores: [93, 91, 92, 96, 98, 100, 95]
6th score: 100  Scores: [96, 92, 91, 93, 98, 100, 95]
6th score: 100  Scores: [96, 92, 91, 97, 94, 100, 95]
6th score: 100  Scores: [96, 92, 97, 91, 94, 100, 95]
6th score: 100  Scores: [91, 93, 92, 96, 98, 100, 95]
6th score: 100  Scores: [92, 96, 91, 93, 98, 100, 95]
6th score: 100  Scores: [92, 96, 91, 97, 94, 100, 95]
6th score: 100  Scores: [92, 96, 97, 91, 94, 100, 95]
```

MiniZinc found a total of 8 solutions, which are listed after the model's code. Later we will see that it is possible to compress the clumsy array sums in the constraint equations, but for now they are left to serve as examples of MiniZinc's array syntax. It should be noted that the include "alldifferent.mzn" statement allows the formation of the "alldifferent" global constraint as required by the problem. This is probably one of the most useful global constraints MiniZinc offers the modeler.

The next example demonstrates the ease with which MiniZinc can solve the interesting, albeit challenging combinatorial problem: number 11 from the from the 2007 American Mathematics Competition 10A Examination:

The numbers from 1 to 8 are placed at the vertices of a cube in such a manner that the sum of the four numbers on each face is the same. What is this common sum?

A)   14   (B)   16   (C)   18   (D)   20   (E)   24

```
include "alldifferent.mzn";

% array for possible vertices values
array[1..8] of var 1..8: a;

%common sum of the four numbers on each face
constraint a[1]+a[2]+a[3]+a[4] = a[7]+a[2]+a[3]+a[6];
constraint a[7]+a[2]+a[3]+a[6] = a[1]+a[5]+a[8]+a[4];
constraint a[1]+a[5]+a[8]+a[4] = a[6]+a[5]+a[8]+a[7];
```

```
constraint a[6]+a[5]+a[8]+a[7] = a[1]+a[2]+a[6]+a[5];
constraint a[1]+a[2]+a[6]+a[5] = a[7]+a[8]+a[3]+a[4];

%6 different numbers
constraint alldifferent(a);

solve satisfy;
output["common sum = "++"\(a[1]+a[2]+a[3]+a[4])"];

% Common sum = 18
```

MiniZinc found a total of 144 different assignments of the numbers 1 to 8 to the vertices of the cube with a constant face sum. Each of these assignments have the common sum of 18.

For our next MiniZinc array example, let us consider problem 25 from the 2019 American Mathematics Competition 10B Examination.

How many sequences of 0 and 1 of length 19 are there that begin with a 0, end with a 0, contain no two consecutive 0s, and contain no three consecutive 1s?

A   55   B   60   C 65   D   70   E   75

```
% use an array to model sequences
array[1..19] of var 0..1: s;

% all sequences begin with 0
constraint s[1] = 0;

% all sequences end with 0
constraint s[19] = 0;

% contain no two consecutive 0s
constraint forall(i in 1..18)(s[i]=1 \/ s[i+1] = 1);

% contain no three consecutive 1s
constraint forall(i in 1..17)(s[i] = 0 \/ s[i+1] = 0 \/ s[i+2] = 0);

solve satisfy;

% 65 solutions

[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
[0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
[0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0])
. . .
[0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0])
[0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0])
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0])
```

MiniZinc found 65 solutions, for which the first and last three are listed above. Often, as with this problem, the total number of solutions are asked for instead of a complete listing of solutions. If one opens the configuration tab and under advanced options selects output statistics for solving then MiniZinc will additionally display the number of solutions found. Even after selecting this option, occasionally MiniZinc will fail to display the total number of solutions. In this case, one can infer the total number from the tally MiniZinc displays.

This model introduces the "forall( )" statement. Just as "for loops" go hand in hand with arrays when programming in imperative languages, the "forall( )" statement is indispensable in forming constraints with arrays in MiniZinc.

As we saw in the map coloring example above, MiniZinc also allows the modeler the ability to create enumerated types. As with the enumerated types found in imperative languages, an enumerated type of size n in MiniZinc behaves much like the integers 1, 2, 3 . . . n. These types can be compared by the order they appear in the definition, act as indices of an array, and be used anywhere in the code where an integer can be used.

For another example of enumerated types in MiniZinc, we consider the Matt Parker puzzle: https://www.think-maths.co.uk/catsanddogs.

```
How many ways can you completely fill your ten kennels, using only cats
and dogs (one animal per kennel), such that no two cats are in adjacent
kennels?

% enumerated data type
enum pets = {cat,dog};

% kennel of size 10
array[1..10] of var pets: kennel;

% no adjacent cats
```

```
constraint forall(i in 1..9)((kennel[i] = cat) -> (kennel[i+1] != cat));

solve satisfy;

% 144 solutions

[cat, dog, cat, dog, cat, dog, cat, dog, cat, dog])
[dog, dog, cat, dog, cat, dog, cat, dog, cat, dog])
[dog, cat, dog, dog, cat, dog, cat, dog, cat, dog]);
. . .
[dog, dog, dog, dog, dog, dog, dog, dog, dog, cat])
[cat, dog, dog, dog, dog, dog, dog, dog, dog, dog])
[dog, dog, dog, dog, dog, dog, dog, dog, dog, dog])
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
matt_p_card_kennel_puzzle.mzn

MiniZinc found 144 total solutions of which the first and last three are shown above. The enumerated type "cat" and "dog" clearly could be represented by a pair of integer values, but their use here allows the model to express transparently how the puzzle is modeled.

Before moving on to sets in MiniZinc, we solve problem 4 from the 2018 American Mathematics Competition 10A Examination. The MiniZinc model presented here illustrates the use of multiple constraints, each using an implies (—>) construct. The last constraint employs the "sum" array function. The sum function is one among other functions that are available in MiniZinc that operate on the arrays:

```
How many ways can a student schedule 3 mathematics courses --
algebra, geometry, and number theory -- in a 6-period day if
no two mathematics courses can be taken in consecutive periods?

(What courses the student takes during the other 3 periods
is of no concern here.)

(A) 3      (B) 6        (C) 12       (D) 18        (E) 24

% D is a dummy or filler course
enum c = {A,G,N,D};

array[1..6] of var c: s;
```

```
% no two mathematics courses can be taken in consecutive periods
constraint forall(d in 1..5)((s[d] = A) -> (s[d+1] != A));
constraint forall(d in 1..5)((s[d] = A) -> (s[d+1] != G));
constraint forall(d in 1..5)((s[d] = A) -> (s[d+1] != N));

constraint forall(d in 1..5)((s[d] = G) -> (s[d+1] != G));
constraint forall(d in 1..5)((s[d] = G) -> (s[d+1] != A));
constraint forall(d in 1..5)((s[d] = G) -> (s[d+1] != N));

constraint forall(d in 1..5)((s[d] = N) -> (s[d+1] != N));
constraint forall(d in 1..5)((s[d] = N) -> (s[d+1] != A));
constraint forall(d in 1..5)((s[d] = N) -> (s[d+1] != G));


% a math course may be scheduled only once
constraint forall(i in 1..5, j in 2..6 where i <j)((s[i] = A) -> (s[j] !=
A));
constraint forall(i in 1..5, j in 2..6 where i <j)((s[i] = G) -> (s[j] !=
G));
constraint forall(i in 1..5, j in 2..6 where i <j)((s[i] = N) -> (s[j] !=
N));

constraint sum(i in 1..6)(bool2int(s[i] = D)) = 3;

solve satisfy;
% display schedules
output[show(s)];

%solutions:     24

[N, D, G, D, A, D]
[G, D, N, D, A, D]
[G, D, D, A, D, N]

[A, D, G, D, D, N]
[G, D, N, D, D, A
[N, D, G, D, D, A]
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%2012/
amc12A_2018_4.mzn

MiniZinc found 24 total solutions for which the first and last three are shown above. We note here again the convenience of the enumerated data type. The use of bool2int (boolean to integer) function in conjunction with the sum function insures exactly three mathematics (or equivalently three filler) courses are scheduled. This problem is an example of a very small size scheduling problem. Scheduling is of vital importance in so many disciplines. The MiniZinc language is ideally suited for modeling scheduling and related problems.

For our last example, we consider Problem 5 from the 2020 American

Mathematics Competition 12A Examination.

> *The 25 integers from -10 to 14 inclusive, can be arranged to form a*
> *5-by-5 square in which the sum of the numbers in each row, the sum of the*
> *numbers in each column, and the sum of the numbers along each of the main*
> *diagonals are all the same.*
>
> *What is the value of this common sum?*
>
> *A 2   B 5   C 10   D 25   E 50*

```
include "alldifferent.mzn";

% 5 x 5 2-D array
array[1..5, 1..5] of var -10..14: g;
var int:c;

%Diagonal conditions
constraint sum (i in 1..5) (g[i,i]) = c;
constraint sum (i in 1..5) (g[i,6-i]) = c;

%Row conditions
constraint sum (j in 1..5) (g[1,j]) = c;
constraint sum (j in 1..5) (g[2,j]) = c;
constraint sum (j in 1..5) (g[3,j]) = c;
constraint sum (j in 1..5) (g[4,j]) = c;
constraint sum (j in 1..5) (g[5,j]) = c;

%Column conditions
constraint sum (i in 1..5) (g[i,1]) = c;
constraint sum (i in 1..5) (g[i,2]) = c;
constraint sum (i in 1..5) (g[i,3]) = c;
constraint sum (i in 1..5) (g[i,4]) = c;
constraint sum (i in 1..5) (g[i,5]) = c;

% insure all 25 numbers used exactly once
constraint alldifferent([g[i,j]|i in 1..5, j in 1..5]);

solve satisfy;

[14, -7, 12, -8, -1, -6, 10, -3, 8, 1, -2, 7, -10, 2, 13, -5, 4, 0, 5, 6,
9, -4, 11, 3, -9]

c = 10
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%2012/
amc12A_2020_5.mzn

MiniZinc found a plethora of solutions to this problem. Only one is listed

above, along with the common sum value: "c = 10". This model employs two-

dimensional arrays, which is another useful data structure available in MiniZinc that is also universally found in other programming languages.

Modeling with Sets

Historically, the concept of a set has been fundamental to mathematics. Sets afford a unifying language for both mathematical definitions and analysis. In accordance, modern languages like Java and Python allow for the formation and manipulation of sets of various data types. MiniZinc singularly allows for sets containing integers to be decision variables. This ability to form constraints on integer valued sets is yet another powerful modeling feature of MiniZinc.

For our first example of modeling with sets, we consider problem 11 from the 1991 American Junior High School Examination:

```
There are several sets of three different numbers whose sum is 15 which
can be chosen from {1,2,3,4,5,6,7,8,9. How many of these sets contain a
5?
(A)3    (B)  4    (C) 5    (D)  6    (E)  7

% form base set
set of int: possible = {1,2,3,4,5,6,7,8,9};

% form all subsets of possible
var set of possible:c;

% sets of three different numbers
constraint card(c)=3;

% sum is 15
constraint sum (i in c) (i) = 15;

% sets contain a 5
constraint 5 in c;

solve satisfy;

c = {1,5,9}
c = {2,5,8}
c = {3,5,7}
c = 4..6;
```

This code above shows how sets are formed in MiniZinc. The card() function is one among others of set operations available for use in constraints. Set membership is expressed with the "in" statement. Lastly, MiniZinc found four

solutions, which are listed after the code. Most likely since MiniZinc represents sets in terms of arrays, the set {4, 5, 6} is displayed as the array 4..6 in the last line.

Similar to the above problem is problem 9 taken from the vast collection of challenging problems provided by Project Euler: https://projecteuler.net/

```
Special Pythagorean triplet

Problem 9
A Pythagorean triplet is a set of three natural numbers, a < b < c, for
which, a^2+ b^2 = c^2.

For example, 3^2 + 4^2 = 9 + 16 = 25 = 5^2..

There exists exactly one Pythagorean triplet for which a + b + c = 1000.

Find the product abc.

% restrict the search
var 1..1000:a;

var int:b;

var int:c;

% b, c depend on a
constraint b in a+1..1000;

constraint c = 1000 - a - b;

% triangle conditions
constraint a < b + c /\ b < a + c /\ c < a + b;

% Pythagorean condition
constraint a*a+b*b=c*c;

% perimeter condition
constraint a+b+c=1000;

solve satisfy;

% display solution
output["product abc = "++"\(a*b*c)"];
```

product abc = 31875000

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
proj_euler_9.mzn

Another set constraint example is problem 12 from the 2004 American

Mathematics Competition 10A Examination:

Henry's Hamburger Heaven offers its hamburgers with the following
*condiments: ketchup, mustard, mayonnaise, tomato, lettuce, pickles,
cheese, and onions. A customer can choose one, two, or three meat
patties, and any collection of condiments.*
How many different kinds of hamburgers can be ordered?

(A)   24   (B)   256   (C)   768   (D)   40,320   (E)   120,960

```
%ingredients: h1, h2, h3, 1, 2, or 3 patties
enum h = {h1, h2, h3, ketchup, mustard, mayonnaise, tomato, lettuce,
pickles, cheese, onions};

%all possible subsets of h
var set of h: possible;

%must have meat patty
constraint card(possible intersect {h1, h2, h3}) = 1;

solve satisfy;

% display sets
output["A burger = "++"\(possible)"];

768 solutions

A burger = {h1, ketchup, mustard, mayonnaise, tomato, lettuce, pickles,
cheese, onions}

A burger = {h1, ketchup, mustard, mayonnaise, tomato, lettuce, pickles,
cheese}

A burger = {h1, ketchup, mustard, mayonnaise, tomato, lettuce, pickles,
onions}
. . .
A burger = {h3, cheese}

A burger = {h3, onions}

A burger = {h3}
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/
AMC%2010/2004_AMC_10A_12.mzn

MiniZinc found 768 total solutions for which the first and last three are shown

above. The enumerated data type is used to specify a hamburger's ingredients.

Note the "intersect" (intersection of two sets) along with the "card" function

constrains any solution set to have exactly one, two, or three hamburger patties.

Since there is only one constraint on possible subsets of {h1, h2, h3, ketchup, mustard, mayonnaise, tomato, lettuce, pickles, cheese, onions} the large number of solutions (768) is not surprising.

Below is yet another Matt Parker puzzle that is ideally suited for solving with sets in MiniZinc: https://www.think-maths.co.uk/primepairs

```
Rearrange the numbers from 1-9, such that all adjacent pairs sum to a
prime number.

include "alldifferent.mzn";


% the numbers from 1-9
par int:n = 9;

% set of possible prime sums
par set of int: primes = {2,3,5,7,11,13,17,19};

% represent solutions in an array
array[1..n] of var 1..n: s;

% adjacent pairs sum to a prime number
constraint forall(i in 1..n-1)((s[i]+s[i+1]) in primes);

% insure each digit is used exactly once
constraint alldifferent(s);

solve satisfy;

% display solutions
output["\(s)"];

% solutions:     140

[7, 4, 9, 8, 5, 6, 1, 2, 3]
[7, 4, 3, 8, 5, 6, 1, 2, 9]
[7, 4, 1, 6, 5, 8, 3, 2, 9]
. . .
[7, 6, 5, 2, 1, 4, 9, 8, 3]
[3, 2, 1, 6, 7, 4, 9, 8, 5]
[7, 6, 1, 2, 3, 4, 9, 8, 5]
```

 https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
matt_p_card_kennel_puzzle.mzn

MiniZinc found 140 solutions for which the first and last three are shown above. This model employs both arrays and sets. Instead of testing if a particular sum pair

is prime, a set of predefined primes {2, 3, 5, 7, 11, 13, 17, 19} serves as a "lookup table" in the constraint to efficiently find prime sum pairs.

With some minor allowances, MiniZinc can occasionally be used to calculate probabilities. Consider problem 12 from the 2018 American Invitational Mathematics Examination 1:

```
For every subset T of U = { 1,2,3,...,18 }, let s(T) be the sum of the
elements of T, with s(empty set) defined to be 0. If T is chosen at
random among all subsets of U, the probability that s(T) is divisible by
3 is m/n, where m and n are relatively prime positive integers. Find m.

% Universal set
set of int:possible = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};

% subsets S of {1,2,3,..,18}
var set of possible:S;

% s(T) is divisible by 3
constraint (sum (i in S) (i)) mod 3 = 0;

solve satisfy;

% Display sets S
output[(show(S))];

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
AIME_1_2018_12.mzn
```

Tallying the results displayed by MiniZinc, one discovers that MiniZinc found a total 87,424 subsets satisfying the question's division criterion. Since the total number of all subsets of an 18 element set is 2^{18} the desired probability is $\dfrac{87,424}{2^{18}}$ which reduces to $\dfrac{683}{2048}$. This yields the correct answer: 683.

Modeling with Comprehensions

List comprehensions originated from the mathematical set-builder notation. They should be familiar to all who have who have programmed in Python. Aside from Python, functional programming languages like Haskell also support list comprehensions. For those who are unfamiliar or might be a bit hazy on set-builder notation, a simple example is given below:

$$S = \{x | x \ is \ an \ int, \ x \geq 0\}$$

This of course describes the integer valued set $\{0,1,2,3,..\}$. MiniZinc's syntax allows for the creation of both list and set comprehensions. Before turning to our first MiniZinc example, we should note that a list or set comprehension is by its nature already a constraint. It should then come as no surprise that comprehensions are a handy construct for the MiniZinc modeler .

For our first example of comprehensions, we consider problem 18 from the 2018 American Mathematics Competition 8 Examination. How many positive factors does 23,232 have?

```
(A) 9      (B) 12      (C) 28      (D) 36      (E) 42
```

```
% solution set
set of int: ans;

% set comprehension
ans = {i | i in 1..23232 where 23232 mod i = 0};

solve satisfy;

% display total number and solution set
output["Total = "++"\(card(ans))"++" "++"\(ans)"];

Total = 42

{1,2,3,4,6,8,11,12,16,22,24,32,33,44,48,64,66,88,96,
121,132,176,192,242,264,352,363,484,528,704,726,968,1056,
1452,1936,2112,2904,3872,5808,7744,11616,23232}
```
https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%208/
amc8_2018_18.mzn

This model employs a set comprehension to specify the solution set. MiniZinc found a total of 42 factors, which are displayed as a set above using the set "card()" function.

A similar set constraint example comes from problem 18 from the 2017 American Mathematics Competition 8 Examination.

```
Let Z be a 6-digit positive integer, such as 247247, whose first three
digits are the same as its last three digits taken in the same order.
Which of the following numbers must also be a factor of Z?

A 11    B 19    C 101    D 111    E 1111

% set of possible Z
set of int: nums;

% first three digits are the same as last three digits
nums = {100000*i+10000*j+1000*k+100*i+10*j+k | i in 1..9, j in 0..9, k in
0..9};

% decision variable
var int:d;

% d limited to the 5 options above
constraint d in {11,19,101,111,1111};

% factor requirement
constraint forall(n in nums)(n mod d = 0);

solve satisfy;

% display solution
output["d = "++" "++"\(d)"];

d =   11

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%208/
amc8_2017_7.mzn
```

This model again employs a set comprehension to restrict a remainder ("mod") constraint. Out of the 5 given multiple choice options, miniZinc determined that the value of d = 11. Note the search is narrowed the by limiting the decision variable "d" to the 5 given options by constraining the variable "d" to the set {11, 19, 101, 111, 1111}. Lastly, we note the problems 6-digit positive integer requirement (first three digits are the same as its last three digits) is concisely modeled with three separate

conditions ("i in 1..9, j in 0..9, k in 0..9") in the set comprehension definition of the set "num".

Another set comprehension problem is 13 from the 2015 American Mathematics Competition 8 Examination.

```
How many subsets of two elements can be removed from the set.
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11} so that the mean (average) of the
remaining numbers is 6?

(A) 1    (B)  2    (C)    3    (D)    5    (E)    6

%Parent set
set of int: possible;
possible = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

%All possible subsets of the parent set
var set of possible:c;

%Require the cardinality of 9
constraint card(c) = 9;

%average of the remaining numbers is 6
%after a two element subset is removed
constraint sum([n|n in c]) = 54;

solve satisfy;

%Display all two element subsets
output["\(possible diff c)"];

%solutions:     5

{5,7}
{4,8}
{3,9}
{2,10}
{1,11}

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%208/
amc8_2015_13.mzn
```

This model starts off by defining the 11 element integer set "possible". The decision variable "c" of all possible subsets is constrained by first requiring that "c" have cardinality of 9 (two elements were removed), and secondly that the average of the remaining elements in "c" equals 54 (6*9 = 54) . This accomplished by applying the set function "sum" to the set comprehension "[n|n in c]". The 5

possible two element subsets are obtained by forming the set difference ("diff")
between the parent set "possible" and the set "c".

The generators of a list comprehension usually do not involve decision
variables. Nonetheless, as demonstrated below list comprehensions can be used to
flatten higher dimensional arrays in constraints.

In combinatorics, a Latin square is defined as an $n \times n$ array filled
with n different symbols, each occurring exactly once in each row and exactly
once in each column. For example the table below is a 3 x 3 Latin square
consisting of the digits $\{1,2,3\}$.

1	2	3
2	3	1
3	1	2

The following MiniZinc code uses list comprehensions to generate a Latin
square of a given size:

```
% Latin Square
include "alldifferent.mzn";

% size of Latin square to be created
int: N = 5;

% 2-D array decision variable
array[1..N,1..N] of var 1..N:sq;

% row condition
constraint forall(i in 1..N)(alldifferent([sq[i,j]|j in 1..N]));

% column condition
constraint forall(j in 1..N)(alldifferent([sq[i,j]|i in 1..N]));

solve satisfy;

% display Latin square
output[show(sq)];
```

```
[3, 1, 2, 4, 5,
 1, 2, 5, 3, 4,
 2, 5, 4, 1, 3,
 4, 3, 1, 5, 2,
 5, 4, 3, 2, 1]
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
latin_sq.mzn

Since the global constraint "alldifferent" operates only on one-dimensional

arrays, a list comprehension is used in both the row and column constraints to

flatten the "sq" array into "N" one-dimensional arrays. One of the many 5 x 5 Latin

squares generated by MiniZinc is listed after the code.

In recreational mathematics, a magic square is a grid of numbers whose row,

column, and diagonal sums are all equal. This common value is usually referred to

as the magic constant. If the square is to be filled with the integers $\{1,2,3,...,n^2\}$

then the magic constant can be seen to equal $\dfrac{n(n^2+1)}{2}$. Consider the model

below, which generates a 5 x 5 magic square:% Magic Square

```
include "alldifferent.mzn";
% size of magic square to be created
int: N = 5;

% 2-D array decision variable
array[1..N,1..N] of var 1..N*N:sq;

% magic constant
int:magic_num = (N*(N*N+1)) div 2;

% list comprehension
constraint alldifferent([sq[i,j]|i in 1..N ,j in 1..N]);

% row condition
constraint forall(i in 1..N)(sum(j in 1..N)(sq[i,j]) = magic_num);

% column condition
constraint forall(j in 1..N)(sum(i in 1..N)(sq[i,j]) = magic_num);

% diagonal conditions
constraint forall(i in 1..N)(sum(i in 1..N)(sq[i,i]) = magic_num);
constraint forall(i in 1..N)(sum(i in 1..N)(sq[N+1-i,i]) = magic_num);

solve satisfy;
```

```
% display magic square
output[show(sq)];

  [25, 5,   9,    3, 23,
   4,  24, 15, 14,   8,
   20, 7,   1, 16, 21,
   6,  17, 18, 13, 11,
   10, 12, 22, 19, 2]
```

Once again since the "alldifferent" global constraint operates only on one-dimensional arrays, a list comprehension is used to flatten the "sq" array into a one-dimensional array. In contrast, the row, column, and diagonal constraints are expressed in terms of the original two-dimensional array. The 5 x 5 magic square with a magic sum of 65 is listed after the code.

We would be amiss if at this point we failed to solve a Sudoku puzzle with a MiniZinc model. It should be noted that a Sudoku puzzle is a special case of a Latin square. Any solution to a Sudoku puzzle is also Latin square. Sudoku puzzles have an additional constraint that in the case of a 9 x 9 grid (the most common Sudoku configuration) all 3 × 3 adjacent sub-squares must contain each of the digits $\{1,2,3,...,9\}$ exactly once. Before turning to a MiniZinc program to solve a Sudoku puzzle, a completed 4 x 4 Sudoku is given below as an example :

We now turn to a bare-bones MiniZinc model, which solves a 4x4 Sudoku puzzle. The model could easily be adapted to solve larger size boards, but since our interest here is in list comprehensions the code presented will suffice. It should be noted that this model was adapted from the MiniZinc tutorial: https://www.minizinc.org/doc-2.4.3/en/modelling2.html#putting-it-all-together

```
include "alldifferent.mzn";

% initial clues board, 0 represents an empty cell
array[1..4,1..4] of 0..4: clues;

% partially filled clue board
clues =  [|0,4,0,0|
           0,0,4,0|
           0,0,0,2|
           1,0,0,0|];

% 4 x 4 board to be solved
array[1..4,1..4] of var 1..4: sudoku;

% initialize sudoku board
constraint forall(i,j in 1..4)(
    if clues[i,j] > 0 then sudoku[i,j] = clues[i,j] else true endif );

% All different in rows
constraint forall (i in 1..4) (
                   alldifferent( [ sudoku[i,j] | j in 1..4 ]) );

% All different in columns.
constraint forall (j in 1..4) (
                   alldifferent( [ sudoku[i,j] | i in 1..4]) );

% All different in sub-squares:
constraint forall (a, o in 1..2)(
                alldifferent( [ sudoku[(a-1) *2 + a1, (o-1)*2 + o1] |
                                a1, o1 in 1..2 ] ) );
solve satisfy;

output  [show(sudoku)];

[3, 4, 2, 1,
 2, 1, 4, 3,
 4, 3, 1, 2,
 1, 2, 3, 4]
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/sudoku.mzn

Here again list comprehensions are used in the last three constraint statements to flatten the two-dimensional array "sudoku". The last constraint

utilizes a second set of inner variables "a1" and "o1". These work in conjunction with the outer variables "a" and "o" to generate each of the four 2 x 2 sub-squares. MiniZinc found a unique solution to the initial "clues" board, which is given after the last line of the model.

For yet another example of list comprehensions, we turn to the classical 8-Queens chess puzzle. This puzzle asks if it is possible to place eight queens on a chessboard so that no two queens are able to attack each other. If one recalls that in chess a queen can move horizontally, vertically, and diagonally then the grid below gives one of the 92 possible solutions to the puzzle:

```
.......Q
.Q......
...Q....
Q.......
......Q.
....Q...
..Q.....
.....Q..
```

 The MiniZinc model that rendered this board is given below: https://www.minizinc.org/doc-2.4.3/en/mzn_search.html

```
include "alldifferent.mzn";

% standard chess board
int: n = 8;

% gives row and column position of queen
% q[2] = 5 col = 2 row = 5
array[1..n] of var 1..n: q;

% no two queens in the same row
constraint all_different(q);


% distinct diagonals
constraint alldifferent([ q[i] + i | i in 1..n]);


% upwards and downwards
constraint alldifferent([ q[i] - i | i in 1..n]);


solve satisfy;
```

```
% Display the board
output [ if fix(q[j]) = i then "Q" else "." endif ++
        if j = n then "\n" else "" endif | i,j in 1..n]
```

Long with their use in the last three constraints, a list comprehension is also employed to render the output.

We conclude this section with the April 2019 IBM Ponder this Puzzle given below. Our MiniZinc model employs both a 2-dimensional array along with a predetermined set of primes to arrive at a solution. By restricting the search to a relatively small set, this "look up table" strategy can often be used in MiniZinc to effectively narrow the search and arrive at a solution in a reasonable amount of time. Of course, there is no guarantee that a solution will be found with such an approach. Nonetheless when solving a puzzle with a seemingly intractable search space, it is usually best to start small and then where possible incrementally enlarge the search space.

```
Find nine different prime numbers that can be placed in a 3x3 square in
such a way that the average of every row, column, and diagonal is also a
prime number.

include "alldifferent.mzn";

% use primes from P
set of int: P = {3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131,
137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199};

% grid to be filled
array[1..3,1..3] of var P:g;

% 8 possible average values
array[1..8] of var P:a;

% row averages are prime
constraint forall(i in 1..3)(sum(j in 1..3)(g[i,j]) = 3*a[I]);

% column averages are prime
constraint forall(j in 1..3)(sum(i in 1..3)(g[i,j]) = 3*a[j+3]);

% diagonal averages are prime
```

```
constraint (sum(i in 1..3)(g[i,i]) = 3*a[7]);

constraint (sum(i in 1..3)(g[i,4-i]) = 3*a[8]);

% nine different prime numbers
constraint alldifferent([g[i,j]|i,j in 1..3]);

solve satisfy;

% Display the grid of 9 different primes

output["\(g[1,1])"++ " "++"\(g[1,2])"++ " "++"\(g[1,3])"++ "\n"++
    "\(g[2,1])"++ " "++"\(g[2,2])"++ " "++"\(g[2,3])"++ "\n"++
    "\(g[3,1])"++ " "++"\(g[3,2])"++ " "++"\(g[3,3])"++ "\n"];
17 29 83
41 5 47
53 23 11

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
IBM_2019.mzn
```

It can be easily checked that 3 x 3 grid of primes found by MiniZinc satisfies

the requirements of the challenge.

Exercises

In what follows, a collection of problems are presented in order to provide the reader modeling practice in MiniZinc. MiniZinc models, which solve each problem can be found at the end of this tract.

1. Find the largest possible value of TWO in the alphametic puzzle:

TWO + TWO = FOUR.

2. Find all solutions to the alphametic puzzle:

NO + MAN + BEARS + SORROW = BETTER (Julius Caesar)

http://www.tkcs-collins.com/truman/alphamet/alphamet.shtml

3. 2015 AIME II Problem 3

Let m be the least positive integer divisible by 17 whose digits sum to 17.

Find m.

4. A stick of wood is to be cut into three unequal pieces. The first piece is shorter than the second piece, and the second is shorter than the third. If the stick is 24 cm long and the length of each piece is an even integer, what are the possibilities for the length of the pieces?

5. Your knapsack has a maximum weight capacity of 104. The weights of 8 items are: [25, 35, 45, 5, 25, 3, 2, 2] and their respective values are: [350, 400, 450, 20, 70, 8, 5, 5]. What items should be selected to maximize the total value you can carry in your knapsack? What is this maximum value?

6. Calculate the maximum allowable loan that can be had for monthly payments of $1700, amortized monthly over 30 years, at an interest

rate of 2.8%.

7. Consider the 8-country map depicted below. Use MiniZinc to find a coloring of the map using only three colors.
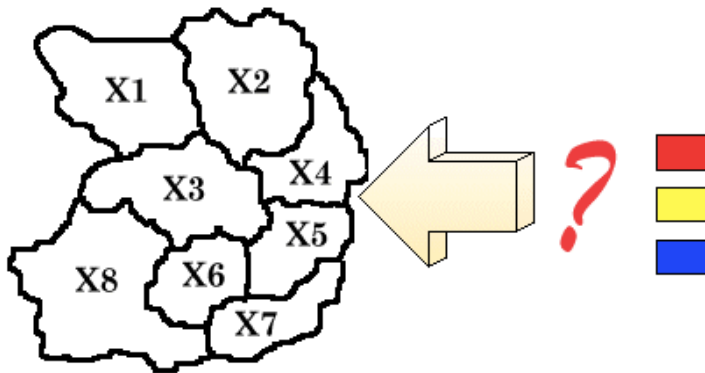
8. Problem 11 2019 AMC 10B

Two jars each contain the same number of marbles, and every marble is either blue or green. In Jar 1 the ratio of blue to green marbles is 9:1, and the ratio of blue to green marbles in Jar 2 is 8:1. There are 95 green marbles in all. How many more blue marbles are in Jar 1 than in Jar 2?

A 5   B 10   C 25   D 45   E 50

9. 2017 AMC 10A Problem 25

How many integers between 100 and 999, inclusive, have the property that some permutation of its digits is a multiple of 11 between 100 and 999? For example, both 121 and 211 have this property.

(A) 226 (B) 243 (C) 270 (D) 469 (E) 486

10. Project Euler Coin sums Problem 31

 https://projecteuler.net/problem=31

In the United Kingdom the currency is made up of pound (£) and pence (p).

There are eight coins in general circulation:

1p, 2p, 5p, 10p, 20p, 50p, £1 (100p), and £2 (200p).

It is possible to make £2 in the following way:

1×£1 + 1×50p + 2×20p + 1×5p + 1×2p + 3×1p

How many different ways can £2 be made using any number of coins?

11. 2006 AMC 12A Problem 25

How many non-empty subsets S of {1,2,3,..,15} have

the following two properties?

(1) No two consecutive integers belong to S.

(2) If S contains k elements, then S contains no number less than k.

(A) 277   (B) 311   (C) 376 (D) 377 (E) 405

12. 2006 AMC 12B Problem 25

A sequence a(n) of non-negative integers is defined by the rule a(n+2) = |
a(n+1)-a(n)| for n >= 1. If a(1) = 999, a(2) < 999 and a(2006) = 1, how many
different values of a(2) are possible?

(A) 165 (B) 324 (C) 495 (D) 499 (E) 660

13. 2004 AMC 10A Problem 22

Let S be the set of the 2005 smallest positive multiples of 4, and let T be the
set of the 2005 smallest positive multiples of 6. How many elements are
common to S and T?

(A) 166 (B) 333 (C) 500 (D) 668 (E) 1001

14. 2010 AMC 12B Problem 17

The entries in a 3 x 3 array include all the digits from 1 through 9,
arranged so that the entries in every row and column are in increasing order.
How many such arrays are there?

15. American High School Mathematics Exam 1994

Nine chairs in a row are to be occupied by 6 students and three professors:
Alpha, Beta, and Gamma. The Professors arrive before the students and
choose their chairs so that each professor will be between two students. In
how many ways can the three professors choose their chairs?

16. Discrete Mathematics Problems

https://www.unf.edu/~wkloster/3100/problems.pdf

Suppose we have 12 different men and 7 different women. How many ways can
we seat them around a circular table so that no two women sit next to each
other? (Note: it may help to assume the seats are numbered)

17. 2015 AIME II Problem 10

Call a permutation $a_1, a_2, \ldots, a_n$ of the integers $1, 2, \ldots, n$ *quasi-increasing* if $a_k \leq a_{k+1} + 2$ for each $1 \leq k \leq n - 1$. For example,
53421 and 14253 are quasi-increasing permutations of the
integers $1, 2, 3, 4, 5$, but 45123 is not. Find the number of quasi-increasing
permutations of the integers $1, 2, \ldots, 7$.

Solutions

1.
```
include "alldifferent.mzn";

% decision variables
var 1..9:T;
var 0..9:W;
var 0..9:O;

var 1..9:F;
var 0..9:U;
var 0..9:R;

% TWO + TWO = FOUR
constraint 2*(100*T + 10*W + O) = 1000*F + 100*O+ 10*U + R;

% unique values for different letters
constraint alldifferent([T, W, O, F, U, R]);

% Maximize TWO
solve maximize (100*T + 10*W + O);

% Display the result
output["TWO = "++"\(100*T + 10*W + O)"++" FOUR = "++"\(1000*F + 100*O+
10*U + R)"];

% TWO = 938 FOUR = 1876
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
two_plus_two.mzn

_____

2.
```
% NO + MAN + BEARS + SORROW = BETTER (Julius Caesar)

include "alldifferent.mzn";

% decision variables
var 1..9:N;
var 0..9:O;

var 1..9:M;
var 0..9:A;

var 1..9:B;
var 0..9:E;
var 0..9:R;
var 1..9:S;

var 0..9:W;

var 0..9:T;


% NO + MAN + BEARS + SORROW = BETTER
constraint 10*N + O + 100*M + 10*A + N + 10000*B +
```

```
            1000*E+ 100*A + 10*R + S + 100000*S + 10000*O+ 1000*R + 100*R
            + 10*O + W = 100000*B + 10000*E+ 1000*T + 100*T + 10*E + R;

constraint alldifferent([N, O, M, A, B, E, R, S, W, T]);

solve satisfy;

output["A = "++"\(A)"++" B = "++"\(B)"++" E = "++"\(E)"++" M = "++"\(M)"
       ++" N = "++"\(B)"++" O = "++"\(O)"++" R = "++"\(R)"
       ++" S = "++"\(S)"++" T = "++"\(T)"++" W = "++"\(W)"];
```

% **A = 1 B = 3 E = 0 M = 8 N = 3 O = 7 R = 4 S = 2 T = 5 W = 9**

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
julius_caesar.mzn

_____

3.
```
% all AIME solutions are three digit numbers
var 0..9:H;
var 0..9:T;
var 0..9:U;

% positive integer
constraint 100*H + 10*T + U > 0;

% digits sum to 17
constraint H + T + U = 17;

% divisible by 17
constraint (100*H + 10*T +U) mod 17 = 0;

% find mininum
solve minimize (100*H + 10*T +U);

output["\(H)"++"\(T)"++"\(U)"];
```

% **476**

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
AIME_II_2015_3.mzn

_____

4.
```
% Stick pieces
var 1..24:P1;
var 1..24:P2;
var 1..24:P3;

% increasing lengths
constraint P1 < P2 /\ P2 < P3;

% even lengths
constraint P1 mod 2 = 0 /\ P2 mod 2 = 0 /\ P3 mod 2 = 0;

% stick is 24 cm long
constraint P1 + P2 + P3 = 24;
```

```
%required solve statement
solve satisfy;

%display results
output["P1 = "++"\(P1)"++" P2 = "++"\(P2)"++" P3 = "++"\(P3)"];
```

MiniZinc found a total of 7 solutions

P1 = 2 P2 = 4 P3 = 18
P1 = 4 P2 = 6 P3 = 14
P1 = 2 P2 = 6 P3 = 16
P1 = 6 P2 = 8 P3 = 10
P1 = 4 P2 = 8 P3 = 12
P1 = 2 P2 = 8 P3 = 14
P1 = 2 P2 = 10 P3 = 12

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
stick_lengths.mzn

_____

5.
```
% 0-1 knapsack
array[1..8] of var 0..1:sack;

% item weights
array[1..8] of int:w = [25, 35, 45, 5, 25, 3, 2, 2];

% item values
array[1..8] of int:v = [350, 400,450, 20, 70, 8, 5, 5];

% total weight <= 104
constraint sum([w[i]*sack[i]|i in 1..8])<=104;

% maximize total sack value
solve maximize(sum([v[i]*sack[i]|i in 1..8]));

% display max value and 0-1 sack values
output["Max value = "++ show(sum([v[i]*sack[i]|i in 1..8]))++" "++
        "sack = "++show(sack)];
```

% **Max value = 900 sack = [1, 0, 1, 1, 1, 0, 1, 1]**

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
eight_knapsack.mzn

_____

6.
```
%Monthly payments
par float:P = 1700.0;

%Interest rate divided over 12
par float:c = 0.028/12; % 2.8 percent divided by 12

%30 years times twelve months
par float:n = 360.0;
```

```
%Loan allowed
var float:L;
%Mortgage equation

constraint P = L*(c*pow(1+c,n))/(pow(1+c,n)-1);

solve satisfy;

%display the result
output["Maximum loan = $"++show_float(6, 2, L)];
```

%**Maximum loan = $413731.56**

_____

7.
```
% Coloring map with 3 colors
enum c = {red,green,blue};

% Country variables
var c: X1;
var c: X2;
var c: X3;
var c: X4;
var c: X5;
var c: X6;
var c: X7;
var c: X8;

% bordering countries have different colors
constraint X1 != X2;
constraint X1 != X3;
constraint X2 != X3;
constraint X2 != X4;
constraint X3 != X4;
constraint X3 != X5;
constraint X3 != X6;
constraint X3 != X8;
constraint X4 != X5;
constraint X5 != X6;
constraint X5 != X7;
constraint X6 != X7;
constraint X6 != X8;
constraint X7 != X8;

solve satisfy;

/*
```
X1 = green;
X2 = blue;
X3 = red;
X4 = green;
X5 = blue;
X6 = green;
X7 = red;
X8 = blue;

```
*/
```

_____

8.

```
% jar variables
var int:J1;
var int:J2;
var int:G1;
var int:B1;
var int:G2;
var int:B2;

% non-zero marbles
constraint J1 >0;
constraint J2 >0;

constraint G1 >0;
constraint B1 >0;

constraint G2 >0;
constraint B2 >0;

% same number of marbles
constraint J1 = J2;
% green plus blue equal total number of marbles
constraint J1 = G1 + B1;
constraint J2 = G2 + B2;
% 95 green marbles in all
constraint G1 + G2 = 95;

%Jar 1 the ratio of blue to green marbles is 9:1
constraint B1 = 9*G1;

%The ratio of blue to green marbles in Jar 2 is 8:1
constraint B2 = 8*G2;

solve satisfy;
output["B2 - B1 = "++"\(B1-B2)"];
```

% **B2 - B1 = 5**

_____

9.

```
% array for possible integers
array[1..3] of var 0..9: d;

% permutation variables
var int:p1;
var int:p2;
var int:p3;
var int:p4;
```

```
var int:p5;
var int:p6;

% permutation of digits
constraint p1 = 100*d[1]+10*d[2]+d[3];
constraint p2 = 100*d[1]+10*d[3]+d[2];
constraint p3 = 100*d[2]+10*d[1]+d[3];
constraint p4 = 100*d[2]+10*d[3]+d[1];
constraint p5 = 100*d[3]+10*d[2]+d[1];
constraint p6 = 100*d[3]+10*d[1]+d[2];

% minimum value is 100
constraint d[1] > 0;

% multiple of 11
constraint p1 mod 11 = 0 \/  p2 mod 11 = 0 \/  p3 mod 11 = 0 \/
p4 mod 11 = 0 \/  p5 mod 11 = 0 \/  p6 mod 11 = 0;

% between 100 and 999
constraint (100 < p1 /\ p1 < 999) \/ (100 < p2 /\ p2 < 999) \/ (100 <
p3 /\ p3 < 999) \/(100 < p4 /\ p4 < 999) \/ (100 < p5 /\ p5 < 999) \/
(100 < p6 /\ p6 < 999);

solve satisfy;
output["\(d)"];
```

% 226 solutions
[1, 1, 0]
[1, 0, 1]
[1, 2, 1]
[1, 1, 2]
[1, 3, 2]
[1, 2, 3]
 . . .
[9, 1, 8]
[9, 6, 8]
[9, 0, 9]
[9, 7, 9]

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/
AMC%2010/2017_AMC_10A_25.mzn

_____

10.

```
% coin variables
var 0..200: p1;
var 0..100: p2;
var 0..40: p3;
var 0..20: p4;
var 0..10: p5;
var 0..4:  p6;
var 0..2:  p7;
var 0..1:  p8;

% coin values
array[1..8] of int:v = [1, 2, 5, 10, 20, 50, 100, 200];
```

```
% sum = 200
constraint v[1]*p1 + v[2]*p2  + v[3]*p3 +  v[4]*p4+
           v[5]*p5 + v[6]*p6  + v[7]*p7 +  v[8]*p8  = 200;

solve satisfy;

output["\([p1, p2, p3, p4, p5, p6, p7, p8])"];

[200, 0, 0, 0, 0, 0, 0, 0]
[198, 1, 0, 0, 0, 0, 0, 0]
[196, 2, 0, 0, 0, 0, 0, 0]
. . .
[4, 98, 0, 0, 0, 0, 0, 0]
[2, 99, 0, 0, 0, 0, 0, 0]
[0, 100, 0, 0, 0, 0, 0, 0]
[ 100 more solutions ]
[ 200 more solutions ]
[ 400 more solutions ]
[ 800 more solutions ]
[ 1600 more solutions ]
[ 3200 more solutions ]
[ 6400 more solutions ]
[ 12800 more solutions ]
[ 25600 more solutions ]
[ 22480 more solutions ]
[0, 0, 0, 0, 0, 0, 0, 1]
```

% **73682 total ways**

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
Euler_31.mzn

_____

11.
```
% Universal set
set of int:possible = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

% subsets S of {1,2,3,..,15}
var set of possible:S;

% non-empty subsets
constraint card(S) > 0;

% (1) No two consecutive integers belong to S.
constraint forall(i,j in S)(abs(i-j) != 1);

% (2) If S contains k elements, then S contains no number less than k.
constraint forall(k in 1..15)(card(S) = k -> forall(i in S)(i>=k));

solve satisfy;

% Display sets S
output[(show(S))];

1..1
{2,4}
{2,5}
. . .
```

```
15..15
```

_____

12.
```
% sequence a(n) non-negative integers
array[1..2006] of var 0..1000:a;

% a(1) = 999,
constraint a[1] = 999;

% a(2) < 999
constraint a[2] < 999;

% a(2006) = 1
constraint a[2006] = 1;

% defined by the rule a(n+2) = |a(n+1)-a(n)|
constraint forall(i in 1..2004)(abs(a[i+1]-a[i]) = a[i+2]);

solve satisfy;

output[show(a[2])];
```

_____

13.
```
% 2005 smallest positive multiples of 4
set of int:S;
S = {4*i | i in 1..2005};

% 2005 smallest positive multiples of 6
set of int:T;
T = {6*i | i in 1..2005};

solve satisfy;

% Display elements common to S and T?
output["Number common = "++"\(card(S intersect T))"];
```

_____

14.
```
include "alldifferent.mzn";
```

```
% 3 x 3 2-D array
array[1..3, 1..3] of var 1..9: g;

% increasing rows
constraint forall(i in 1..3)(g[i,1] < g[i,2] /\ g[i,2] < g[i,3]);

% increasing columns
constraint forall(j in 1..3)(g[1,j] < g[2,j] /\ g[2,j] < g[3,j]);

% include all the digits from 1 through 9
constraint alldifferent([g[i,j]|i in 1..3,j in 1..3]);

solve satisfy;

% Display grids
output["\(g[1,1])"++ " "++"\(g[1,2])"++ " "++"\(g[1,3])"++ "\n"++
      "\(g[2,1])"++ " "++"\(g[2,2])"++ " "++"\(g[2,3])"++ "\n"++
      "\(g[3,1])"++ " "++"\(g[3,2])"++ " "++"\(g[3,3])"++ "\n"];

%  solutions:      42

These are the 3 x 3 symmetric Young tableaux

1 3 7
2 4 8
5 6 9

1 3 6
2 4 8
5 7 9

1 3 5
2 4 8
6 7 9
. . .

1 2 3
4 6 7
5 8 9

1 3 4
2 6 7
5 8 9

1 2 4
3 6 7
5 8 9
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/AMC%2012/
amc12B_2010_17.mzn

_____

15.

```
% S = students A,B,G professors
enum p = {S, A, B, G};

% nine chairs in a row of students and
```

```
array[1..9] of var p:c;


% six students
constraint sum([1| i in 1..9 where c[i] = S]) = 6;

% only one of each professor
constraint sum([1| i in 1..9 where c[i] = A]) = 1;
constraint sum([1| i in 1..9 where c[i] = B]) = 1;
constraint sum([1| i in 1..9 where c[i] = G]) = 1;

% students must be on the ends
constraint c[1] = S /\ c[9] = S;

% each professor will be between two students
constraint forall(i in 2..8)(c[i] = A \/ c[i] = B \/ c[i] = G -> c[i-1] =
S /\ c[i+1] = S);

solve satisfy;

% display seatings
output["valid seating: "++show(c)];

% solutions:      60

valid seating: [S, A, S, G, S, B, S, S, S]
valid seating: [S, A, S, B, S, G, S, S, S]
valid seating: [S, B, S, G, S, A, S, S, S]
. . .
valid seating: [S, G, S, A, S, S, S, B, S]
valid seating: [S, G, S, S, A, S, S, B, S]
valid seating: [S, G, S, S, S, A, S, B, S]
```

_____

16.

```
% M = man W = woman
enum p = {M, W};

% 19 men and women in a circle
array[1..19] of var p:c;

% 12 men
constraint sum([1| i in 1..19 where c[i] = M]) = 12;


% no two women sit next to each other
constraint forall(i in 1..18)(c[i] = W -> c[i+1] = M);

% end of array circle wrap
constraint (c[19] = W -> c[1] = M);

solve satisfy;

% display seatings
output["\(c)"];
```

```
% solutions:      1254

[W, M, W, M, W, M, W, M, W, M, W, M, W, M, M, M, M, M, M]
[M, W, M, W, M, W, M, W, M, W, M, W, M, W, M, M, M, M, M]
[W, M, M, W, M, W, M, W, M, W, M, W, M, W, M, M, M, M, M]
. . .
[M, M, M, W, M, M, W, M, W, M, W, M, W, M, M, W, M, M, W]
[M, W, M, W, M, M, W, M, W, M, M, M, W, M, M, W, M, M, W]
[M, W, M, W, M, M, W, M, M, W, M, M, W, M, M, W, M, M, W]
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
twelve%20_men_seven_women.mzn

_____

17.

```
include "alldifferent.mzn";

% represents the permutation
array[1..7] of var 1..7:a;

% quasi-increasing
constraint forall(i in 1..6)(a[i] <= a[i+1] + 2);

% insures array is a permutation
constraint alldifferent(a);

solve satisfy;

% display all quasi-increasing permutations
output[show(a)];
```

```
% solutions:       486

[7, 5, 4, 3, 2, 1, 6]
[6, 5, 4, 3, 2, 1, 7]
[7, 6, 4, 3, 2, 1, 5]
. . .
[1, 3, 5, 4, 2, 7, 6]
[2, 1, 5, 4, 3, 7, 6]
[1, 2, 5, 4, 3, 7, 6]
```

https://github.com/pjoscely/Math-Comp-minizinc/blob/master/General/
AIME_II_2015_10.mzn