NOPT042 Constraint programming: Tutorial 8 - Global constraints

In [1]:
```
%load_ext ipicat
```

```
Picat version 3.5#5
```

About the constraint `cummulative`

- Picat on GitHub (unofficial)
- Global Constraint Catalog: the cumulative constraint - see the references

The constraint `global_cardinality`

```
global cardinality(List, Pairs)
```

Let `List` be a list of integer-domain variables `[X1, . . ., Xd]`, and `Pairs` be a list of pairs `[K1-V1, . . ., Kn-Vn]`, where each key `Ki` is a unique integer, and each `Vi` is an integer-domain variable. The constraint is true if every element of List is equal to some key, and, for each pair `Ki-Vi`, exactly `Vi` elements of `List` are equal to `Ki`. This constraint can be defined as follows:

```
global_cardinality(List,Pairs) =>
    foreach($Key-V in Pairs)
        sum([B : E in List, B#<=>(E#=Key)]) #= V
    end.
```

---from the guide

Example: Magic sequence

A magic sequence of length n is a sequence of integers $x_0, \ldots, x_{n-1}$ between 0 and $n-1$, such that for all $i \in \{0, \ldots, n-1\}$, the number i occurs exactly x_i-times in the sequence. For instance, 6,2,1,0,0,0,1,0,0,0 is a magic sequence since 0 occurs 6 times in it, 1 occurs twice, etc.

(Problem from the book.)

Let's maximize the sum of the numbers in the sequence.

In [2]:
```
!time picat magic-sequence/magic-sequence.pi 64
```

```
[60,2,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]

real	0m10.022s
user	0m10.008s
sys	0m0.008s
```

In [3]:
```
# !cat magic-sequence/magic-sequence.pi
```

In [4]:
```
!time picat magic-sequence/magic-sequence2.pi 64
!time picat magic-sequence/magic-sequence2.pi 256
```

```
[60,2,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]

real	0m0.044s
user	0m0.030s
sys	0m0.008s
[252,2,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,1,0,0,0]

real	0m1.660s
user	0m1.595s
sys	0m0.065s
```

In [5]:
```
# !cat magic-sequence/magic-sequence2.pi
```

In [6]:
```
!time picat magic-sequence/magic-sequence3.pi 256
!time picat magic-sequence/magic-sequence3.pi 1024
```

```
[252,2,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,1,0,0,0]

real	0m0.117s
user	0m0.089s
sys	0m0.024s
[1020,2,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]

real	0m3.772s
user	0m3.487s
sys	0m0.284s
```

In [7]:
```
# !cat magic-sequence/magic-sequence3.pi
```

The order of constraints

The order might matter to the solver, the above model is an example. When the `cp` solver parses a constraint, it tries to reduce their domains. The implicit (redundant) constranint using `scalar_product` can reduce the model a lot, which is much better to do before parsing `global_cardinality`. (Note: e.g. MiniZinc doesn't preserve the order of constraints during compilation, the behaviour is a bit unpredictable.)

(Heuristic: easy constraints and constraints that are strong (in reducing the search space) should go first??)

The `circuit` constraint

The constraint `circuit(L)` requires that the list L represents a permutation of $1, \ldots, n$ consisting of a single cycle, i.e., the graph with edges $i \to L[i]$ is a cycle. A similar constraint is `subcircuit(L)` which requires that elements for which $L[i] \neq i$ form a cycle.

Example: Knight tour

Given an integer N, plan a tour of the knight on an $N \times N$ chessboard such that the knight visits every field exactly once and then returns to the starting field. You can assume that N is even.

Hint: For a matrix `M` is a matrix, use `M.vars()` to extract its elements into a list.

In [8]:
```
!picat knight-tour/knight-tour.pi 6
```

```
x = {{9,13,7,8,16,10},{15,19,5,18,3,4},{21,1,2,12,6,29},{32,31,25,30,34,11},{14,22,3
5,36,33,17},{27,28,20,26,24,23}}
X:
  9 13  7  8 16 10
 15 19  5 18  3  4
 21  1  2 12  6 29
 32 31 25 30 34 11
 14 22 35 36 33 17
 27 28 20 26 24 23

Tour:
  1 32 29  6  3 18
 30  7  2 19 28  5
 33 36 31  4 17 20
  8 23 34 15 12 27
 35 14 25 10 21 16
 24  9 22 13 26 11
```

In [9]:
```
!cat knight-tour/knight-tour.pi
```

```
/*******************************************************
  Adapted from
  knight_tour.pi
  from Constraint Solving and Planning with Picat, Springer
  by Neng-Fa Zhou, Hakan Kjellerstrand, and Jonathan Fruhman
*******************************************************/
import cp.

main([N]) =>
  N := N.to_int,
  knight(N,X),
  println(x=X),
  println("X:"),
  print_matrix(X),
  extract_tour(X,Tour),
  println("Tour:"),
  print_matrix(Tour).

% Knight's tour for even N*N.
knight(N, X) =>
  X = new_array(N,N),
  X :: 1..N*N,
  XVars = X.vars(),
  % restrict the domains of each square
  foreach (I in 1..N, J in 1..N)
     D = [-1,-2,1,2],
     Dom = [(I+A-1)*N + J+B : A in D, B in D,
             abs(A) + abs(B) == 3,
             member(I+A,1..N), member(J+B,1..N)],
     Dom.length > 0,
     X[I,J] :: Dom
  end,
  circuit(XVars),
  solve([ff,split],XVars).

extract_tour(X,Tour) =>
  N = X.length,
  Tour = new_array(N,N),
  K = 1,
  Tour[1,1] := K,
  Next = X[1,1],
  while (K < N*N)
    K := K + 1,
    I = 1+((Next-1) div N),
    J = 1+((Next-1) mod N),
    Tour[I,J] := K,
    Next := X[I,J]
  end.

print_matrix(M) =>
  N = M.length,
  V = (N*N).to_string().length,
  Format = "% " ++ (V+1).to_string() ++ "d",
  foreach(I in 1..N)
     foreach(J in 1..N)
        printf(Format,M[I,J])
```

```
    end,
    nl
end,
nl.
```

Homework: routing

Vehicle routing is a geralization of the *Traveling Salesman Problem*. There are `N` cities. Their pairwise (integer) distances are given by a matrix `Distance`. We have `K` vehicles. Each vehicle has a depot in one of the cities 1..N. The depots are given by a list `Depots`.

The goal is to plan routes for each vehicle so that each city is visited exactly once and each vehicle returns to its original depot. It is also possible for a vehicle not to move at all. In that case it has visited itsWe want to minimize total distance traveled by the vehicles.

(You can assume that the depot cities are already visited. The route of a vehicle may be empty, if it happens to be optimal.)

Use the constraint `subcircuit` in your model. Running

```
picat routing.pi instance.pi
```

should return the optimal value of `31` and some representation of the route of each vehicle.

In [10]:
```
!cat routing/instance.pi
```

```
instance(N, Distances, K, Depots) =>
    N = 7,
    Distances = {
        { 0, 4, 8,10, 7,14,15},
        { 4, 0, 7, 7,10,12, 5},
        { 8, 7, 0, 4, 6, 8,10},
        {10, 7, 4, 0, 2, 5, 8},
        { 7,10, 6, 2, 0, 6, 7},
        {14,12, 8, 5, 6, 0, 5},
        {15, 5,10, 8, 7, 5, 0}
    },
    K = 2,
    Depots = [2, 5].
```