NOPT042 Constraint programming: Tutorial 4 – Search strategies

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

```
Picat version 3.5#5
```

From last week:

- Solution to the Coin grid problem.
- Best model and solver for the problem? MIP, naturally expressed as an integer program
- Unsatisfiable instances - LP works well.
- For sparse solution sets heuristic approaches may be slow.

Example: N-queens

Place n queens on an $n \times n$ board so that none attack another. How to choose the decision variables?

- How large is the search space?
- Can we use symmetry breaking?
- Consider the *dual* model.

```
In [2]: !time picat queens/queens-primal.pi 8
```

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

real	0m0.015s
user	0m0.000s
sys	0m0.013s
```

```
In [3]: !cat queens/queens-primal.pi
```

```
% n-queens, primal model
import sat.

main([N]) =>
    N := to_int(N),
    queens(N, Q),
    solve(Q),
    if N <= 32 then
        output(Q)
    end.


queens(N, Q) =>
    Q = new_array(N),
    Q :: 1..N,
    all_different(Q),
    all_different([$Q[I] - I : I in 1..N]),
    all_different([$Q[I] + I : I in 1..N]).


output(Q) =>
    N = Q.length,
    foreach(I in 1..N)
        foreach (J in 1..N)
            if Q[I] = J then
                print("Q")
            else
                print(".")
            end
        end,
        print("\n")
    end.
```

In [4]:
```
!time picat queens/queens-dual.pi 64
```

```
*** error(existence_error(mip_solver),solve)


real    0m0.082s
user    0m0.069s
sys     0m0.012s
```

In [5]:
```
!cat queens/queens-dual.pi
```

```
% n-queens, dual model
import mip.

main([N]) =>
    N := to_int(N),
    queens(N, Board),
    solve(Board),
    if N <= 32 then
        output(Q)
    end.


queens(N, Board) =>
    Board = new_array(N, N),
    Board :: 0..1,

    sum([Board[I, J] : I in 1..N, J in 1..N]) #= N,

    % rows
    foreach(I in 1..N)
        sum([Board[I, J] : J in 1..N]) #<= 1
    end,

    % cols
    foreach(J in 1..N)
        sum([Board[I, J] : I in 1..N]) #<= 1
    end,

    % diags
    foreach(K in 1-N..N-1)
        sum([Board[I,J] : I in 1..N, J in 1..N, I-J = K ]) #<= 1
    end,
    foreach(K in 2..2*N)
        sum([Board[I,J] : I in 1..N, J in 1..N, I+J = K ]) #<= 1
    end.


output(Board) =>
    N = Board.length,
    foreach(I in 1..N)
        foreach (J in 1..N)
            if Board[I, J] = 1 then
                print("Q")
            else
                print(".")
            end
        end,
        print("\n")
    end.
```

Sometimes it is best to model the problem in both ways and add *channelling constraints*. (Here it does not help.)

In [6]:
```
!time picat queens/queens-channeling.pi 128
```

```
real	0m3.031s
user	0m2.917s
sys	0m0.096s
```

In [7]:
```
!cat queens/queens-channeling.pi
```

```
% n-queens, primal model
import sat.

main([N]) =>
    N := to_int(N),
    queens(N, Q, Board),
    solve(Q ++ Board),
    if N <= 32 then
        output(Q)
    end.


queens(N, Q, Board) =>
    % primal
    Q = new_array(N),
    Q :: 1..N,
    all_different(Q),
    all_different([$Q[I] - I : I in 1..N]),
    all_different([$Q[I] + I : I in 1..N]),

    % dual
    Board = new_array(N, N),
    Board :: 0..1,
    sum([Board[I, J] : I in 1..N, J in 1..N]) #= N,
    foreach(I in 1..N)
        sum([Board[I, J] : J in 1..N]) #<= 1
    end,
    foreach(J in 1..N)
        sum([Board[I, J] : I in 1..N]) #<= 1
    end,
    foreach(K in 1-N..N-1)
        sum([Board[I, J] : I in 1..N, J in 1..N, I - J = K ]) #<= 1
    end,
    foreach(K in 2..2*N)
        sum([Board[I, J] : I in 1..N, J in 1..N, I + J = K ]) #<= 1
    end,

    % channeling
    foreach(I in 1..N, J in 1..N)
        (Board[I,J] #= 1) #<=> (Q[I] #= J)
    end.


output(Q) =>
    N = Q.length,
    foreach(I in 1..N)
        foreach (J in 1..N)
            if Q[I] = J then
                print("Q")
            else
                print(".")
            end
        end,
        print("\n")
    end.
```

Can the models be improved using symmetry breaking?

Search strategies

And other solver options: see Picat guide (Section 12.6) and the book (Section 3.5)

In [8]:
```
%%picat -n queens
import cp. %try sat, try also mip with the other model

queens(N, Q) =>
    Q = new_array(N),
    Q :: 1..N,
    all_different(Q),
    all_different([$Q[I] - I : I in 1..N]),
    all_different([$Q[I] + I : I in 1..N]).
```

The predicate `time2` also outputs the number of backtracks during the search - a good measure of complexity.

In [9]:
```
%%picat
main =>
    N = 32,
    queens(N, Q),
    time2(solve(Q)).
```

```
CPU time 31.608 seconds. Backtracks: 11461548
```

Which search strategy could work well for our model?

Here's how we can test multiple search strategies (code adapted from the book):

In [10]:
```
%%picat

% Variable selection
selection(VarSels) =>
    VarSels = [backward,constr,degree,ff,ffc,ffd,forward,inout,leftmost,max,min,up]

% Value selection
choice(ValSels) =>
    ValSels = [down,reverse_split,split,up,updown].

main =>
    selection(VarSels),
    choice(ValSels),
    Timeout = 1000, % Timeout in milliseconds
    %Timeout = 10000, % Timeout in milliseconds
    Ns = [64, 128, 256],

    foreach (N in Ns, VarSel in VarSels, ValSel in ValSels)
```

```
        queens(N,Q),
        time2(time_out(solve([VarSel,ValSel], Q),Timeout,Status)),
        println([N,VarSel,ValSel,Status])
    end.
```

```
CPU time 1.001 seconds. Backtracks: 332533

[64,backward,down,time_out]

CPU time 0.999 seconds. Backtracks: 0

[64,backward,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[64,backward,split,time_out]

CPU time 1.979 seconds. Backtracks: 438595

[64,backward,up,time_out]

CPU time 0.993 seconds. Backtracks: 476688

[64,backward,updown,time_out]

CPU time 1.0 seconds. Backtracks: 337963

[64,constr,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[64,constr,reverse_split,time_out]

CPU time 1.0 seconds. Backtracks: 0

[64,constr,split,time_out]

CPU time 1.001 seconds. Backtracks: 202789

[64,constr,up,time_out]

CPU time 0.998 seconds. Backtracks: 432678

[64,constr,updown,time_out]

CPU time 1.001 seconds. Backtracks: 334375

[64,degree,down,time_out]

CPU time 1.0 seconds. Backtracks: 0

[64,degree,reverse_split,time_out]

CPU time 1.0 seconds. Backtracks: 0

[64,degree,split,time_out]

CPU time 0.999 seconds. Backtracks: 192697

[64,degree,up,time_out]
```

```
CPU time 1.001 seconds. Backtracks: 462801

[64,degree,updown,time_out]

CPU time 0.002 seconds. Backtracks: 695

[64,ff,down,success]

CPU time 0.003 seconds. Backtracks: 0

[64,ff,reverse_split,success]

CPU time 0.003 seconds. Backtracks: 0

[64,ff,split,success]

CPU time 0.002 seconds. Backtracks: 382

[64,ff,up,success]

CPU time 0.001 seconds. Backtracks: 115

[64,ff,updown,success]

CPU time 0.002 seconds. Backtracks: 695

[64,ffc,down,success]

CPU time 0.003 seconds. Backtracks: 0

[64,ffc,reverse_split,success]

CPU time 0.003 seconds. Backtracks: 0

[64,ffc,split,success]

CPU time 0.003 seconds. Backtracks: 382

[64,ffc,up,success]

CPU time 0.0 seconds. Backtracks: 115

[64,ffc,updown,success]

CPU time 0.001 seconds. Backtracks: 136

[64,ffd,down,success]

CPU time 0.001 seconds. Backtracks: 0

[64,ffd,reverse_split,success]

CPU time 0.001 seconds. Backtracks: 0

[64,ffd,split,success]
```

```
CPU time 0.001 seconds. Backtracks: 75

[64,ffd,up,success]

CPU time 0.011 seconds. Backtracks: 9120

[64,ffd,updown,success]

CPU time 1.001 seconds. Backtracks: 336291

[64,forward,down,time_out]

CPU time 1.0 seconds. Backtracks: 0

[64,forward,reverse_split,time_out]

CPU time 1.0 seconds. Backtracks: 0

[64,forward,split,time_out]

CPU time 1.001 seconds. Backtracks: 210552

[64,forward,up,time_out]

CPU time 1.001 seconds. Backtracks: 513258

[64,forward,updown,time_out]

CPU time 1.001 seconds. Backtracks: 530675

[64,inout,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[64,inout,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[64,inout,split,time_out]

CPU time 1.643 seconds. Backtracks: 560670

[64,inout,up,time_out]

CPU time 0.997 seconds. Backtracks: 583084

[64,inout,updown,time_out]

CPU time 1.001 seconds. Backtracks: 378308

[64,leftmost,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[64,leftmost,reverse_split,time_out]
```

```
CPU time 1.001 seconds. Backtracks: 0

[64,leftmost,split,time_out]

CPU time 1.001 seconds. Backtracks: 226277

[64,leftmost,up,time_out]

CPU time 1.0 seconds. Backtracks: 523102

[64,leftmost,updown,time_out]

CPU time 1.0 seconds. Backtracks: 501643

[64,max,down,time_out]

CPU time 1.0 seconds. Backtracks: 0

[64,max,reverse_split,time_out]

CPU time 0.001 seconds. Backtracks: 0

[64,max,split,success]

CPU time 0.001 seconds. Backtracks: 53

[64,max,up,success]

CPU time 0.001 seconds. Backtracks: 363

[64,max,updown,success]

CPU time 0.001 seconds. Backtracks: 82

[64,min,down,success]

CPU time 0.0 seconds. Backtracks: 0

[64,min,reverse_split,success]

CPU time 1.001 seconds. Backtracks: 0

[64,min,split,time_out]

CPU time 1.0 seconds. Backtracks: 324636

[64,min,up,time_out]

CPU time 0.02 seconds. Backtracks: 10314

[64,min,updown,success]

CPU time 1.0 seconds. Backtracks: 378346

[64,up,down,time_out]
```

```
CPU time 1.0 seconds. Backtracks: 0

[64,up,reverse_split,time_out]

CPU time 1.0 seconds. Backtracks: 0

[64,up,split,time_out]

CPU time 1.001 seconds. Backtracks: 233825

[64,up,up,time_out]

CPU time 1.001 seconds. Backtracks: 527440

[64,up,updown,time_out]

CPU time 1.0 seconds. Backtracks: 187258

[128,backward,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,backward,reverse_split,time_out]

CPU time 1.0 seconds. Backtracks: 0

[128,backward,split,time_out]

CPU time 1.0 seconds. Backtracks: 119796

[128,backward,up,time_out]

CPU time 1.001 seconds. Backtracks: 245537

[128,backward,updown,time_out]

CPU time 1.001 seconds. Backtracks: 186411

[128,constr,down,time_out]

CPU time 1.0 seconds. Backtracks: 0

[128,constr,reverse_split,time_out]

CPU time 1.0 seconds. Backtracks: 0

[128,constr,split,time_out]

CPU time 1.0 seconds. Backtracks: 111934

[128,constr,up,time_out]

CPU time 1.001 seconds. Backtracks: 233454

[128,constr,updown,time_out]
```

```
CPU time 1.001 seconds. Backtracks: 187059

[128,degree,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,degree,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,degree,split,time_out]

CPU time 1.001 seconds. Backtracks: 111098

[128,degree,up,time_out]

CPU time 1.001 seconds. Backtracks: 234479

[128,degree,updown,time_out]

CPU time 1.001 seconds. Backtracks: 150168

[128,ff,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,ff,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,ff,split,time_out]

CPU time 1.001 seconds. Backtracks: 75971

[128,ff,up,time_out]

CPU time 1.001 seconds. Backtracks: 411381

[128,ff,updown,time_out]

CPU time 1.001 seconds. Backtracks: 153672

[128,ffc,down,time_out]

CPU time 0.997 seconds. Backtracks: 0

[128,ffc,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,ffc,split,time_out]

CPU time 1.001 seconds. Backtracks: 75891

[128,ffc,up,time_out]
```

```
CPU time 1.001 seconds. Backtracks: 421157

[128,ffc,updown,time_out]

CPU time 0.002 seconds. Backtracks: 4

[128,ffd,down,success]

CPU time 0.002 seconds. Backtracks: 0

[128,ffd,reverse_split,success]

CPU time 0.002 seconds. Backtracks: 0

[128,ffd,split,success]

CPU time 0.002 seconds. Backtracks: 3

[128,ffd,up,success]

CPU time 1.002 seconds. Backtracks: 363969

[128,ffd,updown,time_out]

CPU time 1.001 seconds. Backtracks: 186099

[128,forward,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,forward,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,forward,split,time_out]

CPU time 1.001 seconds. Backtracks: 112216

[128,forward,up,time_out]

CPU time 1.001 seconds. Backtracks: 234660

[128,forward,updown,time_out]

CPU time 1.023 seconds. Backtracks: 477755

[128,inout,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,inout,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,inout,split,time_out]
```

```
CPU time 1.001 seconds. Backtracks: 349480

[128,inout,up,time_out]

CPU time 1.001 seconds. Backtracks: 269926

[128,inout,updown,time_out]

CPU time 0.997 seconds. Backtracks: 188312

[128,leftmost,down,time_out]

CPU time 1.002 seconds. Backtracks: 0

[128,leftmost,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,leftmost,split,time_out]

CPU time 1.001 seconds. Backtracks: 112302

[128,leftmost,up,time_out]

CPU time 1.001 seconds. Backtracks: 232381

[128,leftmost,updown,time_out]

CPU time 1.002 seconds. Backtracks: 213669

[128,max,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,max,reverse_split,time_out]

CPU time 1.002 seconds. Backtracks: 0

[128,max,split,time_out]

CPU time 1.001 seconds. Backtracks: 174170

[128,max,up,time_out]

CPU time 1.002 seconds. Backtracks: 252927

[128,max,updown,time_out]

CPU time 1.001 seconds. Backtracks: 243125

[128,min,down,time_out]

CPU time 1.002 seconds. Backtracks: 0

[128,min,reverse_split,time_out]
```

```
CPU time 1.001 seconds. Backtracks: 0

[128,min,split,time_out]

CPU time 1.002 seconds. Backtracks: 135631

[128,min,up,time_out]

CPU time 1.002 seconds. Backtracks: 213831

[128,min,updown,time_out]

CPU time 1.001 seconds. Backtracks: 187967

[128,up,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[128,up,reverse_split,time_out]

CPU time 1.002 seconds. Backtracks: 0

[128,up,split,time_out]

CPU time 1.001 seconds. Backtracks: 113070

[128,up,up,time_out]

CPU time 1.001 seconds. Backtracks: 236682

[128,up,updown,time_out]

CPU time 0.998 seconds. Backtracks: 88809

[256,backward,down,time_out]

CPU time 0.99 seconds. Backtracks: 0

[256,backward,reverse_split,time_out]

CPU time 0.998 seconds. Backtracks: 0

[256,backward,split,time_out]

CPU time 1.001 seconds. Backtracks: 53526

[256,backward,up,time_out]

CPU time 1.002 seconds. Backtracks: 117596

[256,backward,updown,time_out]

CPU time 1.002 seconds. Backtracks: 83895

[256,constr,down,time_out]
```

```
CPU time 1.001 seconds. Backtracks: 0

[256,constr,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[256,constr,split,time_out]

CPU time 1.001 seconds. Backtracks: 50998

[256,constr,up,time_out]

CPU time 1.001 seconds. Backtracks: 110772

[256,constr,updown,time_out]

CPU time 1.002 seconds. Backtracks: 81512

[256,degree,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[256,degree,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[256,degree,split,time_out]

CPU time 1.002 seconds. Backtracks: 102846

[256,degree,up,time_out]

CPU time 1.002 seconds. Backtracks: 117938

[256,degree,updown,time_out]

CPU time 1.001 seconds. Backtracks: 79598

[256,ff,down,time_out]

CPU time 1.001 seconds. Backtracks: 0

[256,ff,reverse_split,time_out]

CPU time 1.001 seconds. Backtracks: 0

[256,ff,split,time_out]

CPU time 0.998 seconds. Backtracks: 37772

[256,ff,up,time_out]

CPU time 1.002 seconds. Backtracks: 157087

[256,ff,updown,time_out]
```

```
CPU time 0.997 seconds. Backtracks: 79073

[256,ffc,down,time_out]

CPU time 0.998 seconds. Backtracks: 0

[256,ffc,reverse_split,time_out]

CPU time 0.998 seconds. Backtracks: 0

[256,ffc,split,time_out]

CPU time 1.002 seconds. Backtracks: 37613

[256,ffc,up,time_out]

CPU time 1.002 seconds. Backtracks: 155722

[256,ffc,updown,time_out]

CPU time 0.022 seconds. Backtracks: 2223

[256,ffd,down,success]

CPU time 0.022 seconds. Backtracks: 0

[256,ffd,reverse_split,success]

CPU time 0.023 seconds. Backtracks: 0

[256,ffd,split,success]

CPU time 0.018 seconds. Backtracks: 1172

[256,ffd,up,success]

CPU time 1.002 seconds. Backtracks: 148440

[256,ffd,updown,time_out]

CPU time 1.001 seconds. Backtracks: 83735

[256,forward,down,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,forward,reverse_split,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,forward,split,time_out]

CPU time 1.002 seconds. Backtracks: 50598

[256,forward,up,time_out]
```

```
CPU time 1.002 seconds. Backtracks: 109939

[256,forward,updown,time_out]

CPU time 1.002 seconds. Backtracks: 264987

[256,inout,down,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,inout,reverse_split,time_out]

CPU time 1.003 seconds. Backtracks: 0

[256,inout,split,time_out]

CPU time 1.002 seconds. Backtracks: 201544

[256,inout,up,time_out]

CPU time 1.002 seconds. Backtracks: 146534

[256,inout,updown,time_out]

CPU time 1.002 seconds. Backtracks: 79578

[256,leftmost,down,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,leftmost,reverse_split,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,leftmost,split,time_out]

CPU time 0.998 seconds. Backtracks: 51149

[256,leftmost,up,time_out]

CPU time 1.002 seconds. Backtracks: 112200

[256,leftmost,updown,time_out]

CPU time 1.002 seconds. Backtracks: 76806

[256,max,down,time_out]

CPU time 0.998 seconds. Backtracks: 0

[256,max,reverse_split,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,max,split,time_out]
```

```
CPU time 1.002 seconds. Backtracks: 86115

[256,max,up,time_out]

CPU time 1.002 seconds. Backtracks: 189740

[256,max,updown,time_out]

CPU time 1.002 seconds. Backtracks: 111573

[256,min,down,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,min,reverse_split,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,min,split,time_out]

CPU time 1.002 seconds. Backtracks: 46072

[256,min,up,time_out]

CPU time 1.002 seconds. Backtracks: 77321

[256,min,updown,time_out]

CPU time 1.002 seconds. Backtracks: 82320

[256,up,down,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,up,reverse_split,time_out]

CPU time 1.002 seconds. Backtracks: 0

[256,up,split,time_out]

CPU time 1.002 seconds. Backtracks: 50133

[256,up,up,time_out]

CPU time 1.002 seconds. Backtracks: 110066

[256,up,updown,time_out]
```

Exercises

Exercise: Magic square

Arrange numbers $1, 2, \ldots, n^2$ in a square such that every row, every column, and the two main diagonals all sum to the same quantity.

- Try to find the best model, solver and search strategy.
- How many magic squares are there for a given n?
- Allow also for a partially filled instance.

Exercise: Minesweeper

Identify the positions of all mines in a given board. Try the following instance (from the book):

```
Instance = {
    {_,_,2,_,3,_},
    {2,_,_,_,_,_},
    {_,_,2,4,_,3},
    {1,_,3,4,_,_},
    {_,_,_,_,_,3},
    {_,3,_,3,_,_}
}.
```

Exercise: Graph-coloring

1. Write a program that solves the (directed) graph 3-coloring problem with a given number of colors and a given graph. The graph is given by a list of edges, each edge is a 2-element list. We assume that vertices of the graph are $1, \ldots, n$ where n is the maximum number appearing in the list.
2. Generalize your program to graph k-coloring where k is a positive integer given on the input.
3. Modify your program to accept the incidence matrix (a 2D array) instead of the list of edges.
4. Add the flag `-n` to output the minimum number of colors (the chromatic number) of a given graph. For example:

```
picat graph-coloring.pi [[1,2],[2,3],[3,4],[4,1]]
picat graph-coloring.pi [[1,2],[2,3],[3,1]] 4
picat graph-coloring.pi "{{0,1,1},{1,0,1},{1,1,0}}" 4
picat graph-coloring.pi -n [[1,2],[2,3],[3,4],[4,1]]
```

Homework: knapsack

There are two common versions of the problem: the general **knapsack** problem:

> Given a set of items, each with a weight and a value, determine **how many of each item** to include in a collection so that the total

> weight is less than or equal to a given limit and the total value is as large as possible.

And the **0-1 knapsack** problem:

> Given a set of items, each with a weight and a value, determine **which items** to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

(In a general knapsack problem, we can take any number of each item, in the 0-1 version we can take at most one of each.)

Example of an instance:

A thief breaks into a department store (general knapsack) or into a home (0-1 knapsack). They can carry 23kg. Which items (and how many of each, in the general version) should they take to maximize profit? There are the following items:

- a TV (weighs 15kg, costs $500),
- a desktop computer (weighs 11kg, costs $350)
- a laptop (weighs 5kg, costs $230),
- a tablet (weighs 1kg, costs $115),
- an antique vase (weighs 7kg, costs $180),
- a bottle of whisky (weighs 3kg, costs $75), and
- a leather jacket (weighs 4kg, costs $125).

This instance is given in the file `data.pi`.

Your goal is to a program for both the problems. The models accept an optional flag "-01" to denote the 0-1 version, and a filename of a data file including the instance. The output should contain the optimal value, and some reasonable representation of the chosen items. (The autograder will only check the presence of the optimum value.)

Running

```
picat knapsack.pi data.pi
```

should output the optimal total value of 2645 and the chosen items `23 of tablet` in some reasonable format. Running

```
picat knapsack.pi -01 data.pi
```

should output the optimal total value of 845 and the chosen items `[tv,laptop,tablet]` in some reasonable format.

Use the solver `cp` (even though `mip` would be better here) and try to find the best model and the best search strategy. You will need to generate larger instances. You can do it in Picat, using the function `random(MinValue, MaxValue) = RandomValue`. See the attached (proof of conceptish) `generate-random-data.pi`.

In [11]:
```
!cat knapsack/data.pi
```

```
instance(Items, Capacity, Values, Weights) =>
    Items = {"tv", "desktop", "laptop", "tablet", "vase", "bottle", "jacket"},
    Capacity = 23,
    Values = {500,350,230,115,180,75,125},
    Weights = {15,11,5,1,7,3,4}.
```