The Assignment

We have to write a Prolog program that can count the number of solutions to forming a pyramid with the SOMA blocks. Meaning we have 7 3D shapes that has to be matched a larger 3D shape.

The Implementation (*source in appendix*)

```
[0,1,1,1,0],
[1,1,3,1,1],
[1,3,7,3,1],
[1,1,3,1,1],
[0,1,1,1,0]
```

Figure 1, the target.

Functionality

I have chosen to implement the target solution as a list of lists of binary values, as seen in figure 1. Coordinates are `(X,Y)`, and the value of the field then indicates which of the 3 blocks are used. The middle is thus `1+2+4 = 7`, indicating it is a solid column.

```
[0,0,0,
 0,3,1,
 0,0,0]
```

Figure 2, 1st block.

The pieces are implemented likewise, only smaller, as seen in figure 2. The pieces are also one long list, as opposed to list of lists, but it makes no practical difference in this program. The implementation represents a 3x3x3 cube, where the center block is always in use since it serves as the pivot for rotations.

Pieces can be rotated around either of the `(X,Y,Z)` axes. Rotating around the X axis is a simple translation from [N1,N2,N3,N4,N5,N6,N7,N8,N9] to [N7,N4,N1,N8,N5,N2,N9,N6,N3] which Prolog handles gracefully. Rotating around either of the `(Y,Z)` axes takes a lot more math, but it is mainly bit-wise operations.

Each of the `rotateN` functions has a `/2` and a `/3` method. The `rotateN/2` performs a single rotation around the axis, whereas `rotateN/3` uses simple recursion to rotate a number of times. Rotating 4 times around a single axis brings the piece back to the initial state, though the program never does more than 3.

To assemble the pieces there are a range of `putPiece` functions. The largest one, `putPiece/9`, merely pass the calls onto `rotatePiece/5` (which itself passes on to various `rotateN/3`) and `putPiece/6`. The latter one is the workhorse starter. It figures out what rows we want to modify, and then passes those rows on to `putPiece/5`, which in turn splits it into columns for `putPiece/4` to handle. Or, we can say it like this: `putPiece/6` handles `Y` (*sic*); `/5` handles `X`; and `/4` handles `Z`.

Then comes the actual search mechanism embodied in the `loopFigure` functions, which is naïve and excruciatingly slow (more on that in the Testing section). This was designed from these plaintext rules:
- For every Piece it has to cycle through (PosX,PosY,PosZ). Meaning, make sure the piece is tried on every coordinate on the target.
- For every combination of (PosX,PosY,PosZ) it has to cycle through (RotX,RotY,RotZ). Meaning, for every coordinate it has to try all possible permutations of the piece.

Starting from the bottom, `loopFigure/1` starts the whole process. This might as well have been `/0`, but I hoped to get some sort of easy output from it. `loopFigure/1` calls `loopFigure/4` which handles looping through the X-position, and calling `loopFigure/5` which handles looping through the Y-position, and calling `loopFigure/6` which handles looping through the Z-position, and calling `loopFigure/7` which handles looping through the X-rotation, and calling `loopFigure/8` which handles looping through the Y-rotation, and calling `loopFigure/9` which handles looping through the Z-rotation. Furthermore, `loopFigure/9` handles adding the extra pieces to the target by calling `loopFigure/4` with them. If we are already on the 7th piece, the current combination of pieces is checked against `isSolution/1` and a '1' is written if it is correct.

This leads to very deep recursion, but it was the only way that seemed possible asides from making a function with 42 arguments.

Optimizations

I added some optimizations in `loopFigure/5` to skip over that recursion branch if we are trying to position a piece in a corner, given that the corners are all 0.
`putPiece/6,/5,/4` has checks to see if we are trying to place a piece beyond the edge of the target.
`putPiece/4` goes further to check if the piece we are currently placing overlaps with the target structure, by a simple Bitwise AND comparison.

I could add more specialized optimizations, but at the risk of making the code even more unreadable and perhaps hindering Prolog's built-in intelligent branching, I figured I'd let Prolog handle the rest. Doing it in this way will create 1:3 times as many results as there really are, since any matching target will also match turned around the X axis up to 3 times.

The Testing Phase

Testing went quite horribly wrong. I have run numerous test with various output to see that it does actually work, and it adds/rotates/moves the pieces correctly...it is, however, slow. I left the finalizing loop running over night, and it had found **4 solutions** (represented as 1111) when I checked on it. A while later I decided to manually terminate it.

For all I can see, it is working correctly, it's just taking a few months to actually find anything.

Output: Main test run

```
1 ?- consult('soma.pl').
% d:/code/prolog/soma.pl compiled 0.00 sec, 0 bytes

2 ?- loopFigure(Res).
1111

Action (h for help) ? abort
% Execution Aborted
```

Result: Manually terminated

Output: Debug run, with output of bad targets

```
...lots more...
[[0, 0, 0, 3, 4], [0, 0, 2, 6, 6], [0, 0, 6, 6, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 0, 7, 6], [0, 0, 1, 7, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 2, 6, 6], [0, 0, 3, 6, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 2, 6, 6], [0, 0, 3, 6, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 2, 6, 6], [0, 0, 3, 6, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 2, 6, 6], [0, 0, 3, 6, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 0, 7, 6], [0, 0, 1, 7, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 2, 6, 6], [0, 0, 6, 6, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 0, 7, 6], [0, 0, 1, 7, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 0, 7, 6], [0, 0, 1, 7, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 2, 6, 6], [0, 0, 6, 6, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 0, 7, 6], [0, 0, 1, 7, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 2, 6, 6], [0, 0, 3, 6, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 2, 6, 6], [0, 0, 6, 6, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 0, 7, 6], [0, 0, 1, 7, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
[[0, 0, 0, 3, 4], [0, 0, 1, 6, 6], [0, 0, 3, 5, 7], [0, 0, 2, 7, 7], [0, 0, 6, 4, 3]]
...lots more...
```

Result: Manually terminated

The Conclusion

Recursion is power, but recursion is slow. It could be that there really only are 4 solutions (or 1, since 3 are rotated versions of it), but I highly doubt that. Nevertheless, I am quite confident that the program will do its job, before the sun burns out.

Source Listing: The Program

```
% Author: Tino Didriksen

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Helper functions, from SWI-Prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%/*
ignore(Goal) :-
        Goal, !.
ignore(_).
%*/
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 3D figure represented in binary 2D.
% A height of 3 blocks is 1+2+4 = 7
isSolution(A) :-
  A == [
    [0,1,1,1,0],
    [1,1,3,1,1],
    [1,3,7,3,1],
    [1,1,3,1,1],
    [0,1,1,1,0]
    ].

% [0,0,0]  rotated X   [0,0,0]          [0,0,0]
% [0,3,1]  clockwise   [0,3,0] again to [1,3,0]
% [0,0,0] would become [0,1,0]          [0,0,0]
% This is an easy translation...
rotateX([N1,N2,N3,N4,N5,N6,N7,N8,N9],
  [N7,N4,N1,N8,N5,N2,N9,N6,N3]).

% A few helper functions for rotateY and rotateZ.
rotate_ceil2north(N1,N2,N3,Result) :-
  (N1 /\ 4 > 0 -> R1 is 1 ; R1 is 0),
  (N2 /\ 4 > 0 -> R2 is 2 ; R2 is 0),
  (N3 /\ 4 > 0 -> R3 is 4 ; R3 is 0),
  Res is R1 + R2 + R3,
  Result = Res.

rotate_mid2mid(N1,N2,N3,Result) :-
  (N1 /\ 2 > 0 -> R1 is 1 ; R1 is 0),
  (N2 /\ 2 > 0 -> R2 is 2 ; R2 is 0),
  (N3 /\ 2 > 0 -> R3 is 4 ; R3 is 0),
  Res is R1 + R2 + R3,
  Result = Res.

rotate_floor2south(N1,N2,N3,Result) :-
  (N1 /\ 1 > 0 -> R1 is 1 ; R1 is 0),
  (N2 /\ 1 > 0 -> R2 is 2 ; R2 is 0),
  (N3 /\ 1 > 0 -> R3 is 4 ; R3 is 0),
  Res is R1 + R2 + R3,
  Result = Res.

% [0,0,0]  rotated Y   [0,0,0]          [0,0,0]
% [0,3,1]  clockwise   [0,2,0] again to [0,6,4]
% [0,0,0] would become [0,2,2]          [0,0,0]
% This is not so easy...
rotateY([N1,N2,N3,N4,N5,N6,N7,N8,N9],Result) :-
  rotate_ceil2north( N1, N4, N7, R1),
  rotate_ceil2north( N2, N5, N8, R2),
  rotate_ceil2north( N3, N6, N9, R3),
  rotate_mid2mid(    N1, N4, N7, R4),
  rotate_mid2mid(    N2, N5, N8, R5),
  rotate_mid2mid(    N3, N6, N9, R6),
  rotate_floor2south(N1, N4, N7, R7),
  rotate_floor2south(N2, N5, N8, R8),
  rotate_floor2south(N3, N6, N9, R9),
  Result = [R1,R2,R3,R4,R5,R6,R7,R8,R9].
```

```
% [0,0,0]  rotated Z   [0,0,0]          [0,0,0]
% [0,3,1]  clockwise   [3,2,0] again to [4,6,0]
% [0,0,0] would become [0,0,0]          [0,0,0]
% Like rotateY, only the other axis.
rotateZ([N1,N2,N3,N4,N5,N6,N7,N8,N9],Result) :-
  rotate_ceil2north( N3, N2, N1, R3),
  rotate_ceil2north( N6, N5, N4, R6),
  rotate_ceil2north( N9, N8, N7, R9),
  rotate_mid2mid(    N3, N2, N1, R2),
  rotate_mid2mid(    N6, N5, N4, R5),
  rotate_mid2mid(    N9, N8, N7, R8),
  rotate_floor2south(N3, N2, N1, R1),
  rotate_floor2south(N6, N5, N4, R4),
  rotate_floor2south(N9, N8, N7, R7),
  Result = [R1,R2,R3,R4,R5,R6,R7,R8,R9].

% Shortcuts for rotating N times.
rotateX(Piece,0,Result) :- Result = Piece.
rotateX(Piece,Num,Result) :-
  Num > 1 -> rotateX(Piece,Res), rotateX(Res,Num-1,Result) ;
  rotateX(Piece,Result).

rotateY(Piece,0,Result) :- Result = Piece.
rotateY(Piece,Num,Result) :-
  Num > 1 -> rotateY(Piece,Res), rotateY(Res,Num-1,Result) ;
  rotateY(Piece,Result).

rotateZ(Piece,0,Result) :- Result = Piece.
rotateZ(Piece,Num,Result) :-
  Num > 1 -> rotateZ(Piece,Res), rotateZ(Res,Num-1,Result) ;
  rotateZ(Piece,Result).

% Helper to rotate a piece on all axis
rotatePiece(Piece,RX,RY,RZ,Result) :-
  rotateX(Piece,RX,PRX), rotateY(PRX,RY,PRY), rotateZ(PRY,RZ,Result).

% Helper for putPiece/5
% This adds a single column together.
% Piece is a column of a piece.
% It checks whether the target Z coordinates are already used, and fails if so.
putPiece(Value,Piece,PosZ,Result) :-
  ((PosZ == 0,Tm is Piece /\ 1,Tm == 0 ->
    PS is Piece >> 1,
    Tn is Value /\ PS,Tn == 0,
    RS is Value + PS,
    Result = RS) ;
  (PosZ == 1 ->
    Tn is Value /\ Piece,Tn == 0,
    RS is Value + Piece,
    Result = RS) ;
  (PosZ == 2,Tm is Piece /\ 4,Tm == 0 ->
    PS is Piece << 1,
    Tn is Value /\ PS,Tn == 0,
    RS is Value + PS,
    Result = RS)).

% Helper for putPiece/6
% This adds the rows together
putPiece([F1,F2,F3,F4,F5],[P1,P2,P3],PosX,PosZ,Result) :-
  ((PosX == 0,P1 == 0 ->
    putPiece(F1,P2,PosZ,R1), putPiece(F2,P3,PosZ,R2),
    Result = [R1,R2,F3,F4,F5]) ;
  (PosX == 1 ->
    putPiece(F1,P1,PosZ,R1),putPiece(F2,P2,PosZ,R2),putPiece(F3,P3,PosZ,R3),
    Result = [R1,R2,R3,F4,F5]) ;
  (PosX == 2 ->
    putPiece(F2,P1,PosZ,R2),putPiece(F3,P2,PosZ,R3),putPiece(F4,P3,PosZ,R4),
    Result = [F1,R2,R3,R4,F5]) ;
  (PosX == 3 ->
    putPiece(F3,P1,PosZ,R3),putPiece(F4,P2,PosZ,R4),putPiece(F5,P3,PosZ,R5),
    Result = [F1,F2,R3,R4,R5]) ;
  (PosX == 4,P3 == 0 ->
    putPiece(F4,P1,PosZ,R4), putPiece(F5,P2,PosZ,R5),
    Result = [F1,F2,F3,R4,R5])).
```

```
% (PosX,PosY) ranges from (0,0) to (4,4)
putPiece([F1,F2,F3,F4,F5],[N1,N2,N3,N4,N5,N6,N7,N8,N9],PosX,PosY,PosZ,Result) :-
  % Add them to the figure
  ((PosY == 0,N1 == 0,N2 == 0,N3 == 0 ->
    putPiece(F1,[N4,N5,N6],PosX,PosZ,R1),
    putPiece(F2,[N7,N8,N9],PosX,PosZ,R2),
    Result = [R1,R2, F3,F4,F5]) ;
  (PosY == 1 ->
    putPiece(F1,[N1,N2,N3],PosX,PosZ,R1),
    putPiece(F2,[N4,N5,N6],PosX,PosZ,R2),
    putPiece(F3,[N7,N8,N9],PosX,PosZ,R3),
    Result = [R1,R2,R3, F4,F5]) ;
  (PosY == 2 ->
    putPiece(F2,[N1,N2,N3],PosX,PosZ,R2),
    putPiece(F3,[N4,N5,N6],PosX,PosZ,R3),
    putPiece(F4,[N7,N8,N9],PosX,PosZ,R4),
    Result = [F1, R2,R3,R4, F5]) ;
  (PosY == 3 ->
    putPiece(F3,[N1,N2,N3],PosX,PosZ,R3),
    putPiece(F4,[N4,N5,N6],PosX,PosZ,R4),
    putPiece(F5,[N7,N8,N9],PosX,PosZ,R5),
    Result = [F1,F2, R3,R4,R5]) ;
  (PosY == 4,N7 == 0,N8 == 0,N9 == 0 ->
    putPiece(F4,[N1,N2,N3],PosX,PosZ,R4),
    putPiece(F5,[N4,N5,N6],PosX,PosZ,R5),
    Result = [F1,F2,F3, R4,R5])).

% All in one shortcut
putPiece(Figure,Piece,RotX,RotY,RotZ,PosX,PosY,PosZ,Result) :-
  rotatePiece(Piece,RotX,RotY,RotZ,Rotated),
  putPiece(Figure,Rotated,PosX,PosY,PosZ,Result).

% The Pieces
%    1       2       3       4       5       6       7
% [0,0,0] [0,0,0] [0,0,0] [0,0,0] [0,1,1] [0,0,0] [0,0,0]
% [0,3,1] [0,7,1] [0,7,2] [0,3,6] [0,3,0] [1,3,0] [0,3,1]
% [0,0,0] [0,0,0] [0,0,0] [0,0,0] [0,0,0] [1,0,0] [0,1,0]

% For every Piece it has to cycle through (PosX,PosY).
% For every combination of (PosX,PosY) it has to cycle through (RotX,RotY,RotZ)
loopFigure([F1,F2,F3,F4,F5],Piece,PosX,PosY,PosZ,RotX,RotY,RotZ,[R1,R2,R3,R4,R5]) :-
  (RotZ < 4,
  RZ is RotZ+1,
  loopFigure([F1,F2,F3,F4,F5],Piece,PosX,PosY,PosZ,RotX,RotY,RZ,[R1,R2,R3,R4,R5])) ;
  ((
    TempFig = [F1,F2,F3,F4,F5],
    TempRes = [R1,R2,R3,R4,R5],
    putPiece(TempFig,Piece,RotX,RotY,RotZ,PosX,PosY,PosZ,TempRes) ->
    (
      (Piece == [0,0,0,0,3,1,0,0,0] ->
        loopFigure(TempRes,[0,0,0,0,7,1,0,0,0],0,Res)) ;
      (Piece == [0,0,0,0,7,1,0,0,0] ->
        loopFigure(TempRes,[0,0,0,0,7,2,0,0,0],0,Res)) ;
      (Piece == [0,0,0,0,7,2,0,0,0] ->
        loopFigure(TempRes,[0,0,0,0,3,6,0,0,0],0,Res)) ;
      (Piece == [0,0,0,0,3,6,0,0,0] ->
        loopFigure(TempRes,[0,1,1,0,3,0,0,0,0],0,Res)) ;
      (Piece == [0,1,1,0,3,0,0,0,0] ->
        loopFigure(TempRes,[0,0,0,1,3,0,1,0,0],0,Res)) ;
      (Piece == [0,0,0,1,3,0,1,0,0] ->
        loopFigure(TempRes,[0,0,0,0,3,1,0,1,0],0,Res)) ;
      (Piece == [0,0,0,0,3,1,0,1,0] ->
        isSolution(TempRes), write('1'))
    )
  )).

loopFigure(Figure,Piece,PosX,PosY,PosZ,RotX,RotY,Result) :-
  (RotY < 4,
  RY is RotY+1,
  loopFigure(Figure,Piece,PosX,PosY,PosZ,RotX,RY,Result)) ;
  loopFigure(Figure,Piece,PosX,PosY,PosZ,RotX,RotY,0,Result).

loopFigure(Figure,Piece,PosX,PosY,PosZ,RotX,Result) :-
  (RotX < 4,
  RX is RotX+1,
  loopFigure(Figure,Piece,PosX,PosY,PosZ,RX,Result)) ;
  loopFigure(Figure,Piece,PosX,PosY,PosZ,RotX,0,Result).

loopFigure(Figure,Piece,PosX,PosY,PosZ,Result) :-
  (PosZ < 3,
  PZ is PosZ+1,
  loopFigure(Figure,Piece,PosX,PosY,PZ,Result)) ;
  loopFigure(Figure,Piece,PosX,PosY,PosZ,0,Result).
```

```
loopFigure(Figure,Piece,PosX,PosY,Result) :-
  (PosX == 0,PosY == 0 -> loopFigure(Figure,Piece,0,1,Result)) ;
  (PosX == 0,PosY == 4 -> loopFigure(Figure,Piece,1,Result)) ;
  (PosX == 4,PosY == 0 -> loopFigure(Figure,Piece,4,1,Result)) ;
  (PosX == 4,PosY == 4 -> loopFigure(Figure,Piece,4,4,4,4,4,4,Result)) ;
  ((PosY < 5,
  PY is PosY+1,
  loopFigure(Figure,Piece,PosX,PY,Result)) ;
  loopFigure(Figure,Piece,PosX,PosY,0,Result)).

loopFigure(
                [[F11,F12,F13,F14,F15],[F21,F22,F23,F24,F25],[F31,F32,F33,F34,F35],[F41,F42,F43,F44,F45],
[F51,F52,F53,F54,F55]],
 Piece,PosX,Result) :-
     Figure   =   [[F11,F12,F13,F14,F15],[F21,F22,F23,F24,F25],[F31,F32,F33,F34,F35],[F41,F42,F43,F44,F45],
[F51,F52,F53,F54,F55]],
  ((PosX < 5,
  PX is PosX+1,
  loopFigure(Figure,Piece,PX,Result)) ;
  loopFigure(Figure,Piece,PosX,0,Result)).

loopFigure(Result) :-
  loopFigure([[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0]],
  [0,0,0,0,3,1,0,0,0],0,Result).
```