Introduction to Linear Programming

Steven Miller -- Fall 2014

```
?LinearProgramming
```

LinearProgramming[c, m, b] finds a vector x that minimizes the quantity $c.x$ subject to the constraints $m.x \geq b$ and $x \geq 0$.
LinearProgramming[c, m, {{b_1, s_1}, {b_2, s_2}, …}] finds a vector x that minimizes $c.x$ subject to $x \geq 0$ and linear constraints specified by the matrix m and the pairs {b_i, s_i}. For each row m_i of m, the corresponding constraint is $m_i.x \geq b_i$ if $s_i == 1$, or $m_i.x == b_i$ if $s_i == 0$, or $m_i.x \leq b_i$ if $s_i == -1$.
LinearProgramming[c, m, b, l] minimizes $c.x$ subject to the constraints specified by m and b and $x \geq l$.
LinearProgramming[c, m, b, {l_1, l_2, …}] minimizes $c.x$ subject to the constraints specified by m and b and $x_i \geq l_i$.
LinearProgramming[c, m, b, {{l_1, u_1}, {l_2, u_2}, …}] minimizes $c.x$ subject to the constraints specified by m and b and $l_i \leq x_i \leq u_i$.
LinearProgramming[c, m, b, lu, dom] takes the elements of x to be in the domain dom, either Reals or Integers.
LinearProgramming[c, m, b, lu, {dom_1, dom_2, …}] takes x_i to be in the domain dom_i. >>

Diet Problem : 10 x + 4 y >= 9, 5 x + 8 y >= 11, 3 x + 2 y >= 5, minimize 2 x + 3 y

In[26]:=
```
(* Here is how we do a simple linear
 programming problem if real variables and sizes small *)
(* we first enter the matrix A, where we enter row by row *)
(* we then enter the b-vector,
which must have the same number of entries as rows of A *)
(* we then enter the cost vector c,
which must have the same number of entries as the columns of A *)
(* the way the code works is we solve Ax ≥ b, x ≥ 0, min(c.x) *)
A = {{10, 4}, {5, 8}, {3, 2}};
b = {9, 11, 5};
c = {2, 3};
soln = LinearProgramming[c, A, b];
Print["The solution is ", soln]
Print["The cost is ", c.soln]
Print["Now let's do it where we require integer values."];
intsoln = LinearProgramming[c, A, b, Automatic, Integers];
Print["The solution in integers is ", intsoln]
Print["The integer solution cost is ", c.intsoln]
```

The solution is $\left\{\frac{9}{7}, \frac{4}{7}\right\}$

The cost is $\frac{30}{7}$

Now let's do it where we require integer values.

LinearProgramming::lpip : Warning: integer linear programming will use a machine–precision approximation of the inputs. >>

The solution in integers is {1, 1}

The integer solution cost is 5

Chess Problem: n queens on n x n board ....

In[3]:=
```
(* making this a program so can call it with different arguments easily *)
(* have a board that is boardsize by boardsize and placing numberofqueens *)
(* use underscore to indicate a variable *)
(* Module indicates module / procedure / function,
the {} means no local variables *)
(* Thus all output can be felt / seen later *)
(* if printlinearprogrammingconditions is 1 print the matrix A and rest,
else don't *)
pawnqueenproblem[boardsize_, numberofqueens_,
  printlinearprogrammingconditions_] := Module[{},
  n = boardsize; (*just change for different board *)
  numqueens = numberofqueens;
  (* allows the number of queens to differ from board size *)
  (* when programming you want freedom -- see
   what matters and what depends on what *)
  (* in some of the constraints what matters is the board size,
  and in others its the  number of queens *)
  (* variables x1, ...., xn, x_{n+1} = x_{2,1}, ... *)
  (* followed by p1, ..., pn, p_{n+1}, ... *)
  (* the above requires some explanation. We
   can't have our variables having indices unfortunately *)
  (* we need a linear list of variables, not an array *)
  (* fortunately it's very simple to pass from indices to a linear list *)
  (* if we have x_{ij} with 1 ≤ i, j ≤  n then we let num(i,j) = (i-1)*n + j *)
  (* note that counts from 1 to n^2 *)
  (* we have two variables, the x_ij,
  which say whether or not a queen is at (i,j), and *)
```

```
(* p_ij, which say whether or not we can place a pawn at (i,j). *)
(* so x_ij --> x_num(i,j) is 1 if a queen is at (i,j) and 0 otherwise *)
(* and p_ij -->
 x_{n^2 + num(i,j)} is 1 if a pawn is at (i,j) and 0 otherwise *)
(* notice that we are always using x subscript
 an integer for our variables. *)
(* we write the queen variables first and then the pawn variables,
and hence adding n^2 in the subscript *)

A = {}; (* initializes our constraint matrix A to be empty,
we'll add constraints *)
(* notice we don't want to type all the constraints by hand,
but write code to add *)
bvec = {}; (* initializes bvector to empty *)

(* the next lines take into account
 which squares on the chessboard can attack (i,j) *)
(* we first record which squares can have a queen attacking (i,j),
remembering we count linearly and must convert *)
(* numbers to pairs; thus if we want to investigate what happens
  for a queen at (a,b) that corresponds to index (a-1)*n + b *)
(* while similarly we could go from inex (a-1)*n + b to (a,b);
there is some small issue with how Mathematica looks at remainders *)
(* and so we change b if it is 0 mod n to n. *)
(* the constraint is the following:
  -Sum_{(a,b) attacks (i,j)} x_{ab} - numqueens p_{ij} ≥ - numsqueens *)
(* this is NOT my first choice for how to write the constraint,
but remember Mathematica does Ax ≥ b *)
(* eventually we will maximize the sum of p_{ij},
so if it is available for a pawn the program will place one there *)
(* (of course we maximize the sum of p_{ij} by minimizing the sum of -p_{ij},
 as we work with minimums!) *)
(* returning to the constraint: if all the x_{ab} that can
   attack (i,j) are zero then we may take p_ij to be 1 *)
(* if even one x_{ab} is 1 then the constraint is too negative on
 the LHS if p_ij is 1, and thus we have p_ij = 0 as desired *)
For[i = 1, i ≤ n, i++,
 For[j = 1, j ≤ n, j++,
  (* the i and j for statements go over all board locations *)
  {
   temp = {}; (* initialize temp to be empty,
   we'll start putting the constraint info here and then append to A *)
   For[num = 1, num ≤ n^2, num++, (* remember we index variables linearly,
    this goes over the n^2 squares that can attack (i,j) *)
    {
     b = Mod[num, n];
     If[b == 0, b = n];
     a = ((num - b) / n) + 1; (* this converts from the linear index
```

```
        to a pair (a,b) for the board space under consideration *)
       (*numat = (a-1)*n + b;*) (* moves from (a,b) to num from 1 to n^2 *)
       (* we now see if a queen at (a,b) could attack
        square (i,j). there are four ways this could happen. *)
       (* they could have the same x-coord, so i = a;
       they could have the same y-coord, so j = b. *)
       (* they could also be on the same upward sloping diagonal,
       so i-j = a-b, or downward, so i-j = a-b *)
       (* the || is how we do an or condition; if either of the four
         conditions hold we append a -1 to our list, else a 0 *)
       (* this will give us n^2 elements, the first half of a row for A;
       we then do the p_ij entries *)
       temp = AppendTo[temp, If[a == i || b == j ||
            i - j == a - b || i + j == a + b, -1, 0]];
      }]; (* end of nuum loop *)
     numij = (i - 1) * n + j;
     (* this gives the linear index corresponding to (i,j) *)
     (* we now finish the row constraint; we put a -
      numqueens at the place corresponding to (i,j) and a 0 elsewhere *)
     For[num = 1, num ≤ n^2, num++, temp =
       AppendTo[temp, If[num == numij, -numqueens, 0]]];
     A = AppendTo[A, temp]; (* add constraint to A *)
     bvec = AppendTo[bvec, -numqueens]; (* add the entry to bvector *)
    }]; (* end of j loop *)
 ]; (* end of i loop *)

 (* now we want to add constraints saying there are EXACTLY numqueen queens,
 ie, numqueen of the xij are 1 and the rest are 0 *)
 (* as Mathematica does Ax ≥ b we need two constraints to get an
   equality: one blah ≥ numqueens and one -blah  ≥ -numqueens. *)
 (* we initialize temp to be zero, and put a 1 in the first n^2
  elements (those corresponding to xij) and a 0 elsewhere *)
 (* we then put a numqueens for the entry of b-vec;
 this gives the constraint sum xij ≥ numqueens *)
 (* the next lines are similar and give -sum x_ij ≥ -numqueens *)
 temp = {}; (* always remember to reinitialize
  the temp list to empty before adding things to it! *)
 For[num = 1, num ≤ 2 n^2, num++, temp = AppendTo[temp, If[num ≤ n^2, 1, 0]]];
 A = AppendTo[A, temp]; (* now we add our constraint to the A matrix *)
 bvec = AppendTo[bvec, numqueens];
 temp = {};
 For[num = 1, num ≤ 2 n^2, num++, temp = AppendTo[temp, If[num ≤ n^2, -1, 0]]];
 A = AppendTo[A, temp];
 bvec = AppendTo[bvec, -numqueens];
 (* this finishes making sure we have numqueen queens *)

 (* annoyingly I could only find commands for
  Mathematica to do integer programming, NOT binary programming *)
```

```
(* we can declare the variables to be integers but not 0,
1 integers. fortunately this is easily fixed *)
(* we just need to add a constraint that each variable is at most 1,
as they are assumed to be at least 0 (advantages canonical form!) *)
(* As the constraints are always Ax ≥ b,
if we want x ≤ 1 we have to program that as -x ≥ -1 *)
(* for each of the 2n^2 variables, the n^2 choices of xij and the n^2 of pij,
we make sure it is at most 1 *)
temp = {}; (* always initialize to empty *)
For[num = 1, num ≤ 2 n^2, num++, (* go through the 2n^2 variables *)
 {
  temp = {}; (* for each variable choice make our list empty,
  put a -1 in the right spot and 0's elsewhere *)
  For[counter = 1, counter ≤ 2 n^2, counter++, AppendTo[temp,
    If[counter == num, -1, 0]]]; (* if in right spot 1, else 0 *)
  A = AppendTo[A, temp]; (* appends new constraint to A,
  and then next line appends -1 as needed to b-vector *)
  bvec = AppendTo[bvec, -1];
 }];

(* now we deine the vector needed for the optimization. we
 initiallize it to empty, and then make it the right size *)
(* since we can only do minima we use -1 as the entries for c
  corresponding to the pawn variable locations, and 0 for the queen *)
c = {};
For[num = 1, num ≤ 2 n^2, num++, c = AppendTo[c, If[num > n^2, -1, 0]]];
(* appending information to c *)

(* below is the key line -- it calls the linear program solver,
and the last bits inform it that the variables are integers *)
(* we save the output to a quantity we named soln,
for solution; we will then format the answer nicely *)
soln = LinearProgramming[c, A, bvec, Automatic, Integers];
Print["Solution is ", soln]; (* prints the soln vector,
but hard to parse so we work on it a bit *)

queenlist = {}; (* initializes the list of queens to empty,
this is the first n^2 variables *)
(* we then go through the soln list and save that info to the queen list *)
(* the next lines after this redo
 this and make a list of pawn solution information *)
(* we really don't need to do this -- we can work directly with
 the solution list, but thought this might be easier to parse *)
For[num = 1, num ≤ Length[soln], num++,
 If[num ≤  n^2, queenlist = AppendTo[queenlist, soln[[num]]]]];
pawnlist = {};
For[num = 1, num ≤ Length[soln], num++,
 If[num > n^2, pawnlist = AppendTo[pawnlist, soln[[num]]]]];
```

```
(* now we will draw a board and place
 Q for queen at P for pawn at the correct locations *)
board = {}; (* as always initialize to empty, this will be a matrix *)
For[i = 1, i ≤ n, i++, (* will construct the board row by row *)
 {
  temp = {}; (* initialize new row of matrix to empty and will add *)
  For[j = 1, j ≤ n, j++,  (* go through the n elements of the row *)
   {
    numat = (i - 1) * n + j;
    (* convert from (i,j) board location to linear index number *)
    If[queenlist[[numat]] == 1, temp = AppendTo[temp, "Q"],
     (* if queen there write Q *)
     If[pawnlist[[numat]] == 1, temp = AppendTo[temp, "P"],
      (* if pawn there write P *)
      temp = AppendTo[temp, "-"]]]; (* else write -
     to show space empty but show the space is there *)
   }];
  board = AppendTo[board, temp]; (* save the new row to the board matrix *)
 }];
Print[MatrixForm[board]]; (* print the board matrix NICELY *)
Print["Number of pawns = ", Sum[pawnlist[[i]], {i, 1, Length[pawnlist]}]];
(* prints number pawns *)
Print["Number of queens = ", Sum[queenlist[[i]], {i, 1, Length[queenlist]}]];
(* prints number queens *)
If[printlinearprogrammingconditions == 1,
 {
  Print["Constraint Matrix is"];
  Print[MatrixForm[A]]; (* prints the constraint matrix *)
  Print["b-vector is"];
  Print[bvec]; (* prints the b-vector *)
  Print["c vector for optimization is"];
  Print[c] ;(* prints the c vector *)
 }]; (* end of print condition --
 only print if printlinearprogrammingconditions is 1 *)
]; (* end of module *)
```

In[4]:=
```
pawnqueenproblem[1, 1, 0]
```

LinearProgramming::lpip : Warning: integer linear programming will use a machine–precision approximation of the inputs. >>

```
Solution is {1, 0}
( Q )
Number of pawns = 0
Number of queens = 1
```

In[5]:=
```
pawnqueenproblem[2, 2, 0]
```

LinearProgramming::lpip : Warning: integer linear programming will use a machine–precision approximation of the inputs. >>

```
Solution is {1, 1, 0, 0, 0, 0, 0, 0}

( Q  Q )
( -  - )

Number of pawns = 0

Number of queens = 2
```

In[6]:= **`pawnqueenproblem[3, 3, 0]`**

LinearProgramming::lpip : Warning: integer linear programming will use a machine–precision approximation of the inputs. >>

```
Solution is {0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

( -  -  Q )
( -  -  - )
( Q  Q  - )

Number of pawns = 0

Number of queens = 3
```

In[18]:= **`Timing[pawnqueenproblem[4, 4, 0]]`**

LinearProgramming::lpip : Warning: integer linear programming will use a machine–precision approximation of the inputs. >>

```
Solution is
 {1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0}

( Q  -  Q  - )
( -  -  -  Q )
( -  P  -  - )
( -  -  -  Q )

Number of pawns = 1

Number of queens = 4
```

Out[18]= {0.093601, Null}

In[20]:= **`Timing[pawnqueenproblem[5, 5, 1]]`**

LinearProgramming::lpip : Warning: integer linear programming will use a machine–precision approximation of the inputs. >>

```
Solution is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0,
  1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

( -  P  -  -  P )
( -  -  -  -  P )
( Q  -  -  -  - )
( Q  -  -  Q  - )
( -  -  Q  Q  - )

Number of pawns = 3

Number of queens = 5

Constraint Matrix is

( -1 -1 -1 -1 -1 -1 -1  0  0  0 -1  0 -1  0  0 -1  0  0 -1  0 -1  0  0  0 -1 -[illegible]
  -1 -1 -1 -1 -1 -1 -1 -1  0  0  0 -1  0 -1  0  0 -1  0  0 -1  0 -1  0  0  0  0
  -1 -1 -1 -1 -1  0 -1 -1 -1  0 -1  0 -1  0 -1  0  0 -1  0  0  0  0 -1  0  0  0
  -1 -1 -1 -1 -1  0  0 -1 -1 -1  0 -1  0 -1  0 -1  0  0 -1  0  0  0  0 -1  0  0
  -1 -1 -1 -1 -1  0  0  0 -1 -1  0  0 -1  0 -1  0 -1  0  0 -1 -1  0  0  0 -1  0
  -1 -1  0  0  0 -1 -1 -1 -1 -1 -1 -1  0  0  0 -1  0 -1  0  0 -1  0  0 -1  0  0
  -1 -1 -1  0  0 -1 -1 -1 -1 -1 -1 -1 -1  0  0  0 -1  0 -1  0  0 -1  0  0 -1  0
   0 -1 -1 -1  0 -1 -1 -1 -1 -1  0 -1 -1 -1  0 -1  0 -1  0 -1  0  0 -1  0  0  0
```

0	0	−1	−1	−1	−1	−1	−1	−1	−1	0	0	−1	−1	−1	0	−1	0	−1	0	−1	0	0	−1	0	0
0	0	0	−1	−1	−1	−1	−1	−1	−1	0	0	0	−1	−1	0	0	−1	0	−1	0	−1	0	0	−1	0
−1	0	−1	0	0	−1	−1	0	0	0	−1	−1	−1	−1	−1	−1	−1	0	0	0	−1	0	−1	0	0	0
0	−1	0	−1	0	−1	−1	−1	0	0	−1	−1	−1	−1	−1	−1	−1	−1	0	0	0	−1	0	−1	0	0
−1	0	−1	0	−1	0	−1	−1	−1	0	−1	−1	−1	−1	−1	0	−1	−1	−1	0	−1	0	−1	0	−1	0
0	−1	0	−1	0	0	0	−1	−1	−1	−1	−1	−1	−1	−1	0	0	−1	−1	−1	0	−1	0	−1	0	0
0	0	−1	0	−1	0	0	0	−1	−1	−1	−1	−1	−1	−1	0	0	0	−1	−1	0	0	−1	0	−1	0
−1	0	0	−1	0	−1	0	−1	0	0	−1	−1	0	0	0	−1	−1	−1	−1	−1	−1	−1	0	0	0	0
0	−1	0	0	−1	0	−1	0	−1	0	−1	−1	−1	0	0	−1	−1	−1	−1	−1	−1	−1	−1	0	0	0
0	0	−1	0	0	−1	0	−1	0	−1	0	−1	−1	−1	0	−1	−1	−1	−1	−1	0	−1	−1	−1	0	0
−1	0	0	−1	0	0	−1	0	−1	0	0	0	−1	−1	−1	−1	−1	−1	−1	−1	0	0	−1	−1	−1	0
0	−1	0	0	−1	0	0	−1	0	−1	0	0	0	−1	−1	−1	−1	−1	−1	−1	0	0	0	−1	−1	0
−1	0	0	0	−1	−1	0	0	−1	0	−1	0	−1	0	0	−1	−1	0	0	0	−1	−1	−1	−1	−1	0
0	−1	0	0	0	0	−1	0	0	−1	0	−1	0	−1	0	−1	−1	−1	0	0	−1	−1	−1	−1	−1	0
0	0	−1	0	0	0	0	−1	0	0	−1	0	−1	0	−1	0	−1	−1	−1	0	−1	−1	−1	−1	−1	0
0	0	0	−1	0	−1	0	0	−1	0	0	−1	0	−1	0	0	0	−1	−1	−1	−1	−1	−1	−1	−1	0
−1	0	0	0	−1	0	−1	0	0	−1	0	0	−1	0	−1	0	0	0	−1	−1	−1	−1	−1	−1	−1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	0
−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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	−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

$$
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

```
b-vector is

{-5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5, -5,
 -5, -5, -5, -5, -5, -5, 5, -5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}

c vector for optimization is

{0, 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, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
```

Out[20]= `{0.436803, Null}`

In[21]:= **`Timing[pawnqueenproblem[6, 6, 0]]`**

LinearProgramming::lpip : Warning: integer linear programming will use a machine-precision approximation of the inputs. >>

```
Solution is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0,
  0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

$$
\begin{pmatrix}
- & P & - & - & P & - \\
P & - & - & - & P & - \\
- & P & - & - & - & - \\
- & - & - & Q & - & Q \\
- & - & Q & - & - & - \\
- & - & Q & Q & - & Q
\end{pmatrix}
$$

```
Number of pawns = 5

Number of queens = 6
```

Out[21]= `{5.148033, Null}`

In[22]:= **`Timing[pawnqueenproblem[7, 7, 0]]`**

LinearProgramming::lpip : Warning: integer linear programming will use a machine-precision approximation of the inputs. >>

```
Solution is
 {1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

$$\begin{pmatrix} Q & - & - & - & - & - & Q \\ - & - & P & P & P & - & - \\ Q & - & - & - & - & - & Q \\ - & - & P & - & P & - & - \\ - & - & - & P & - & P & - \\ - & Q & - & - & - & - & - \\ Q & - & - & - & - & - & Q \end{pmatrix}$$

```
Number of pawns = 7

Number of queens = 7
```

Out[22]= {138.248086, Null}

In[23]:= **Timing[pawnqueenproblem[8, 8, 0]]**

LinearProgramming::lpip : Warning: integer linear programming will use a machine−precision approximation of the inputs. >>

```
Solution is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0,
  0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 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}
```

$$\begin{pmatrix} - & - & - & P & P & P & - & - \\ P & - & - & - & P & P & - & - \\ P & P & - & - & - & P & - & - \\ P & P & - & - & - & - & - & - \\ - & - & - & - & - & - & Q & - \\ - & - & - & - & - & - & Q & Q \\ - & - & Q & - & - & - & Q & Q \\ - & - & Q & - & - & - & - & Q \end{pmatrix}$$

```
Number of pawns = 11

Number of queens = 8
```

Out[23]= {2759.158487, Null}

In[24]:= **2759 / 60.**

Out[24]= 45.9833