NqProblemExtended application.

Description : This program was born to play around the N Queens Problem.
We give a solution which considers the problem from a whole different aspect and splits the actual calculation from the real rules of the chess game. In this way, the calculations of rooks and bishops are also available as well as their super or awesome versions.
(super means that the pieces can also attack as a knight and)
(awesome means the above but till the edge of the chessboard)
Preplaced chess pieces are also available to solve N Queens Completion problem.

Published : 01.01.2018

Current version : 1.0

Developed by : Jozsef Kiss
<thehobbypianist@gmail.com>

Change log : 1.0 - 01.01.2018
Initial release.

Before and during the implementing of the java application, there was no research work to not be guided by other thoughts.
The main goal was the creation of a unique logic and working solution!
The thinking was the classical: to count the possible placings to the N size board, the pieces have to be placed in all of the correct ways and this positions have to be counted (found++).
It was not a goal to notice elementary or complex rules between the placings to "save" several other placings and to save less or more processing time.
According to this, the actual found placing can be printed immediately and without any further calculation on every found++ events.

The base thought was that if a quicker than classical recursive-backtrack solution exists then it has to be simple or not too complex because the running time of that program will be necessarily longer otherwise.
So, the alternative solution has to be able to be found in a finite but rather a "short" time.

The main goals were the followings:
1. the running time of the searching solutions on a n=15 size board has to be the half of the original solution which is 1:44s -> has to be under 52s!
   (the original solution will also be implemented to compare these elapsed times on the same computer)
2. to place every queens, no cheating! :)
3. if possible, to find a solution which contains the logic of the searching and the rules of the chess game separately
4. to use basic datatypes to port this software to another languages easily
5. the time of this project is maximum 1 month gross time

It can be seen below that the aboves are successfully achieved:
1. the 15 queens can be searched (~17s) under the sixth of the time of searching on the original way (1:44s)
2. every queens will be placed
3. anything is possible to search on the board by the separation of the chess rules and the logic of searching and we declared some new chess pieces just for fun
   and, for example bishop is also searchable which is a tricky chess piece to find its placings on the classical way.
4. ints, Strings, booleans, arrays will be used
   to write onto the console and to get the system time, it will use the java.lang.System package which can be replaced easily to others
5. it is done but all of the developing, implementing and testing works are done in almost a quarter year as outside of main work hours

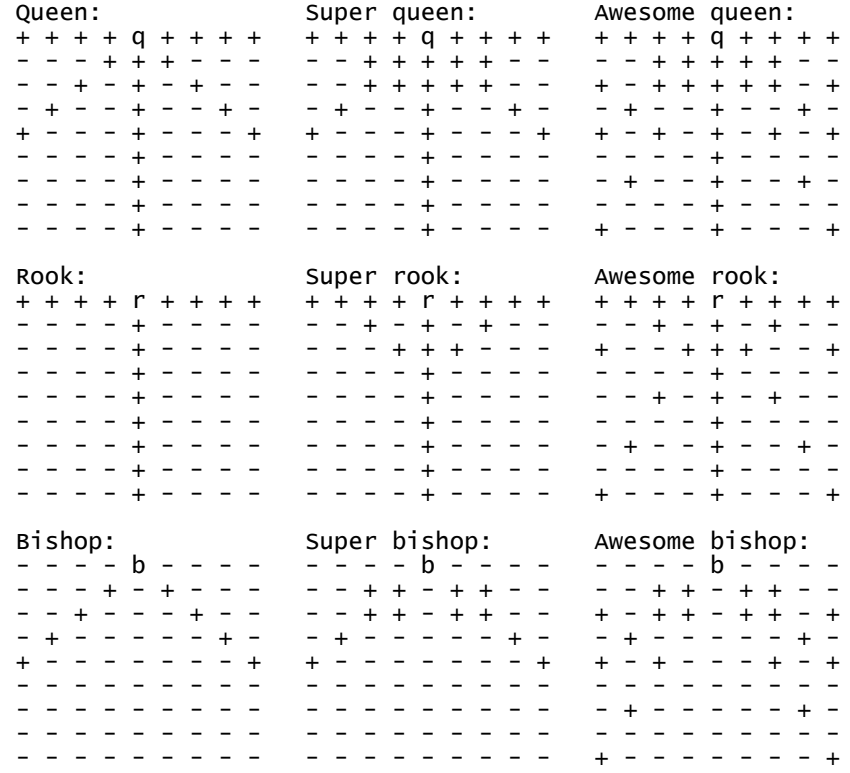
+1. the logic was born to find all (the placing order matters) and the ordered (the ordered solution matters only) placings with a minimal conversion

+2. this software gives a 9 in 1 solution, 9 kinds of chess pieces are searchable

+3. preplaced chess pieces are available to specify so the N Queens (Figures) Completion problem is also solvable.

Almost everyone wanted to put queens onto the chessboard because this piece can hit a lot of directions. If we don't count the pieces that can step only one at a time: pawns and king, there are other chess pieces to attack on the whole chessboard:
 - queen
 - rook
 - bishop
These chess pieces can attack till the edge of the chessboard. The knight seems to be forgotten.
We know the kind super queen. This is a queen that can attack according to the classical rules of the chess game and also to L directions. In this way, we can define this property to all of the aboves:
 - super queen
 - super rook
 - super bishop
So, we can use the knight too. But why should we stop here?
The ability of attacking till to the edge of the chessboard is known so we can declare the awesome property similar to the super property. The difference is: the super chess piece can attack only 1 knight distance while the awesome piece can do the same but till the edge of the chessboard in that direction.
We can use the followings:
 - awesome queen
 - awesome rook
 - awesome bishop

The possible attackings of all of the aboves are:



Applying the aboves, this 9 chess figures can be placed by using this program.

The used solution.

The basic concept is: let's think forward and not backward!
Calculate the usable positions immediately after a piece placing. We can achieve that the next placing will happen to an exactly correct place. This methodology will use less resources to calculate, we guess.
(Instead of the regular solutions which place a queen into a position and check whether other already placed pieces can attack that or not.)

We will represent the chessboard using an ordered sequence.
Fe.4x4:
0 1 2 3   -> 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
4 5 6 7
8 9 10 11   The representation of an N dimensional chessboard is an ordered
12 13 14 15   sequence from 0 to (N^2 - 1).

If we place a chess piece, the following will happen.
0: we have an input sequence that may be the original or a subset of that
1: we will choose an element of this input sequence
2: we calculate the sub-sequence of the input according to the aboves and
   the precalculated attacking map

It can be guaranteed that the next piece placing will happen onto an exactly correct place.

From this, all of the solutions will be found, even including the orders of the placed pieces.
If we want to know only one of these then we have to choose one.
This will be the ordered positions of that solutions.
Fe. (4x4): we will find
1 7 8 14, 1 7 14 8, 1 8 7 14, 1 8 14 7, 1 14 7 8, 1 14 8 7 ...24 * 2 = 48 solutions.
If we would like to know the different solutions then we will catch only the ordered ones, so we will have 1 7 8 14 and 2 4 11 13.
To achieve that, the precalculated attacking map will be changed. This would be enough but we can use some optimization in the logic of calculation to speed up the searchings.
(The right positions will be found according only to the attacking map
  but there will be too many avoidable dead searches.)

Options.

```
  Mode: [ o , i , t ]
    o: original solution, using backtrack-recursion
       it will run a little bit quicker because it uses the precalculated attacking map
    i: improved solution that can place all of the above mentioned 9 chess pieces
    t: testing mode to compare the running times of original and improved
  Dimension: [ int ]
    the chessboard should be the size of this and this number of chess pieces
    have to be placed onto that
( the other options can be applied to the improved mode )
  Pieces: [ q , r , b ]
    q: queen
    r: rook
    b: bishop
  Kinds: [ r , s , a ]
    r: regular, chess piece able to attack as the traditional
    s: super, traditional plus knight attackings
    a: awesome, traditional plus knight attackings till the edge
  Hits: [ o , a , f ]
    o: ordered puts count
    a: all of the puts will be searched for, that means, the orders of the
       pieces will appear:
       all solutions == dimension! * ordered solutions
    f: first placing will be found
  Uniques: [ y , n ]
    y: counts the solutions which cannot be rotated into previously found
       ( it is slow because there have to be extra efforts to find a rotated
         version already placed of the current )
    n: every found placings count and we don't search for already placed
       and rotated versions
  Log: [ n , i , d ]
    n: no log
    i: info, prints the number of found solutions and the elapsed times by
       the placed pieces into the first row
    d: debug, every information will be printed to the console during the searching
  Placing: [ int values separated by  ; character ]
    these chess figures have to be placed initially, can be leaved as empty
```

The core algorithm will be copied into several places because of the many options, so the size of the source code is relatively large.

The core algorithm to run i mode:

0. precalculate the elements of isFiltered 2 dimensional array by the given rule set according to:
   - dimension (n)
   - what pieces will be used (q , r , b)
   - what kind of those (r , s , a)
   - hits (o , a , f)
1. filter function which gives what elements of the input sequence can be used further by the currently placed piece
   applyFiltersXXX (2):

```
    boolean [ ] a = isFiltered [ currPiecePos ] ;
    for ( int i = from ; i < to ; i ++ )
    {
      if ( ! a [ workingArray [ i ] ] )
      {
        workingArray [ currIndToWrite ] = workingArray [ i ] ;
        currIndToWrite ++ ;
      }
    }
    return currIndToWrite - to ;
```

2. recursive method to put the pieces onto the next position from the previous filtering
   putPiecesXXX (18):

```
    if ( pieceToPlace < dimension )
    {
      if ( to - from >= dimension - pieceToPlace )
      {
        int count ;
        for ( int i = from ; i < to ; i ++ )
        {
          currPath [ pieceToPlace ] = workingArray [ i ] ;
          currIndToWrite = to ;
          count = applyFilters ( currPath [ pieceToPlace ] , from , to ) ;
          putPiecesXXX ( pieceToPlace + 1 , currIndToWrite-count , currIndToWrite ) ;
        }
      }
      else
      {
        deads ++ ;
      }
    }
    else
    {
      found ++ ;
    }
```

8-10 modifications of the above algorithm have been tried (for example: 2 pieces have been placed at the same time or the pieces have been placed onto the midline, etc.), but none of these could run faster. We found that the running time is the minimum when the above principle is implemented in this way.

The curves to fit mostly to the run results of the aboves:

D	elapsed (ms)
13	536
14	2941
15	17580
16	117029
17	820977
18	6081743

5th polynomial:
y = -17692570000 + 5994815000*x - 811125900*x^2 + 54785620*x^3 - 1847318*x^4 + 24879.31*x^5

exponential:
y = 0.289471 + 1.398972e-9*e^(+2.000462*x)

The results of using this software.

We have calculated some cases and the results of these can be seen below.

The used configuration was
 - Windows 10 x64 OS
 - Intel Celeron N2840 (2 logical CPU cores, 2.16GHz)
 - 8GB 1333MHz DDR3

Calculating awesome queens up to 28 is an exception because other computer
has been used. The parameters are mentioned there.

////////// T E S T I N G M O D E //////////

This is the running of test (t) mode which compares the method
original (o) and improved (i).

The command was:
  java -jar NqProblemExtended.jar t 18

mode o -> mode i (improved version versus original solution)
mode o won't be run,
  the times have to be written manually to the source
rate1: i elapsed / o elapsed
rate2: i (k) elapsed / i (k-1) elapsed

```
n   count      elapsed      elapsed2             -> elapsed       elapsed2           count        rate1   rate2
13  c73712     2377     ms ( 2s 377ms )         -> 536       ms ( 536ms )          c73712     ( 0.225 |       )
14  c365596    15116    ms ( 15s 116ms )        -> 2941      ms ( 2s 941ms )       c365596    ( 0.194 | 5.486 )
15  c2279184   104357   ms ( 1m 44s 357ms )      -> 17580     ms ( 17s 580ms )      c2279184   ( 0.168 | 5.977 )
16  c14772512  766915   ms ( 12m 46s 915ms )     -> 117029    ms ( 1m 57s 29ms )    c14772512  ( 0.152 | 6.656 )
17  c95815104  6040290  ms ( 1h 40m 40s 290ms )  -> 820977    ms ( 13m 40s 977ms )  c95815104  ( 0.135 | 7.015 )
18  c666090624 48165728 ms ( 13h 22m 45s 728ms ) -> 6081743   ms ( 1h 41m 21s 743ms ) c666090624 ( 0.126 | 7.407 )
```

////////// A W E S O M E Q U E E N S O N 4 0 T H R E A D S //////////

This calculation was made on a laptop, having parameters:
 - Dell 3521
 - Intel i3-3227 (4 logical CPU cores, 1.9GHz)
 - 2x4 GB 1333MHz DDR3

The thread pool has been used on 40 threads because the running time is the
lowest in this case.

The command of the last calculation was:
  java -jar NqProblemExtended.jar i 28 q a o 40 n i

dimension	solutions	elapsed
1	1	47ms
2 -> 9	0	47 -> 64ms
10	4	78ms
11	33	78ms
12	6	78ms
13	59	79ms
14	8	94ms
15	12	109ms
16	18	187ms
17	180	250ms
18	124	594ms
19	361	1s 687ms
20	516	6s 860ms
21	689	25s 728ms
22	2092	2m 0s 498ms
23	5639	8m 32s 988ms
24	22794	41m 52s 252ms
25	68044	3h 9m 59s 430ms
26	275732	15h 45m 42s 275ms
27	767820	3d 3h 3m 47s 476ms
28	3698242	17d 10h 58m 14s 423ms

awesome queens on 40 threads

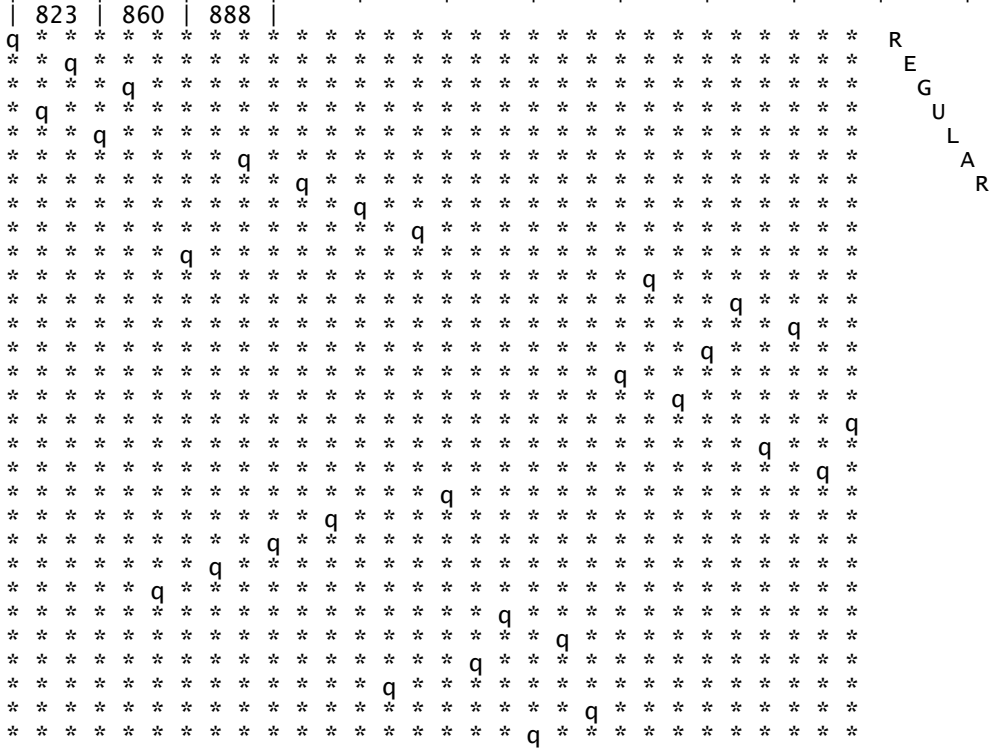
////////// D E B U G M O D E O F 4 Q U E E N S //////////

Let's see an example of using debug mode!
This case will display the core logic, the input and output sequences will be
shown according to the currently placed queen and the rule set (by queen).

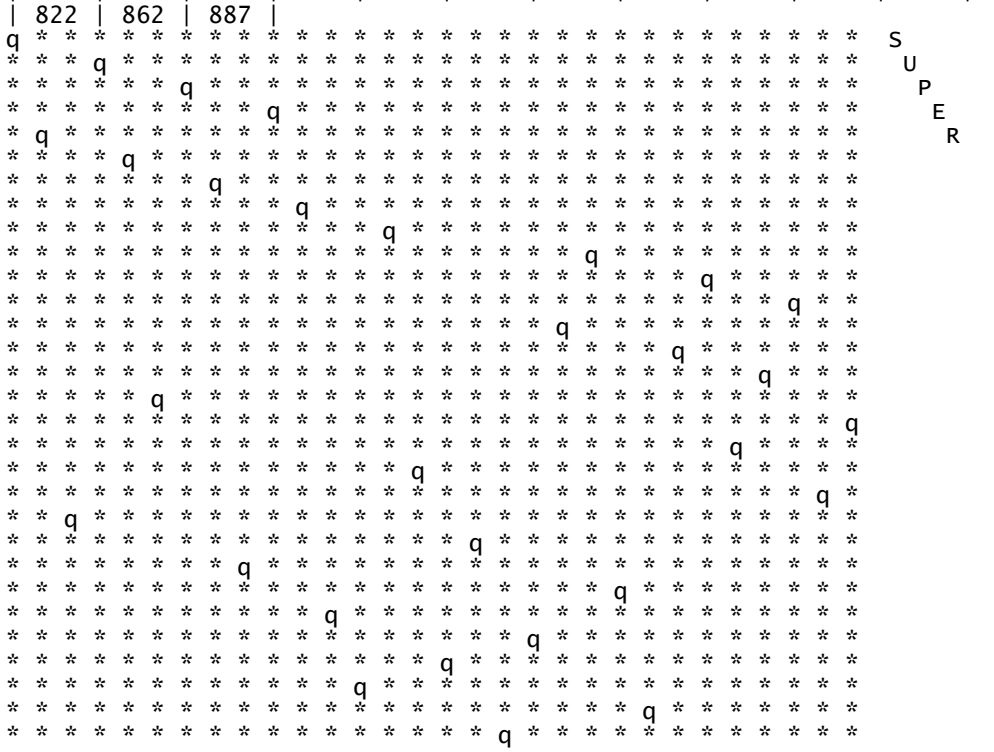
Command:
  java -jar NqProblemExtended.jar i 4 q r o 1 n d

```
  [ ]
                              | ----------+-------------------------------------
                              | to filter : [ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
                              |             , 9 , 10 , 11 , 12 , 13 , 14 , 15 ]
                              | Piece     : 0
                              | filtered  : [ 6 , 7 , 9 , 11 , 13 , 14 ]
  [ 0   ]
                              | ----------+-------------------------------------
                              | to filter : [ 6 , 7 , 9 , 11 , 13 , 14 ]
                              | Piece     : 6
                              | filtered  : [ 13 ]
  [ 0   6   ]
        0
                              | ----------+-------------------------------------
                              | to filter : [ 6 , 7 , 9 , 11 , 13 , 14 ]
                              | Piece     : 7
                              | filtered  : [ 9 , 14 ]
  [ 0   7   ]
                              | ----------+-------------------------------------
                              | to filter : [ 9 , 14 ]
                              | Piece     : 9
                              | filtered  : [ ]
  [ 0   7   9   ]
            0
        0
    0
                              | ----------+-------------------------------------
                              | to filter : [ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
                              |             , 9 , 10 , 11 , 12 , 13 , 14 , 15 ]
                              | Piece     : 1
                              | filtered  : [ 7 , 8 , 10 , 12 , 14 , 15 ]
  [ 1   ]
                              | ----------+-------------------------------------
                              | to filter : [ 7 , 8 , 10 , 12 , 14 , 15 ]
                              | Piece     : 7
                              | filtered  : [ 8 , 12 , 14 ]
  [ 1   7   ]
                              | ----------+-------------------------------------
                              | to filter : [ 8 , 12 , 14 ]
                              | Piece     : 8
                              | filtered  : [ 14 ]
  [ 1   7   8   ]
                              | ----------+-------------------------------------
                              | to filter : [ 14 ]
                              | Piece     : 14
                              | filtered  : [ ]
  [ 1   7   8   14  ]

| 1 | 7 | 8 | 14 |
* q * *
* * * q
q * * *
* * q *
            1
        1
        1
    1
                              | ----------+-------------------------------------
                              | to filter : [ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
                              |             , 9 , 10 , 11 , 12 , 13 , 14 , 15 ]
                              | Piece     : 2
                              | filtered  : [ 4 , 9 , 11 , 12 , 13 , 15 ]
  [ 2   ]
                              | ----------+-------------------------------------
                              | to filter : [ 4 , 9 , 11 , 12 , 13 , 15 ]
                              | Piece     : 4
                              | filtered  : [ 11 , 13 , 15 ]
  [ 2   4   ]
                              | ----------+-------------------------------------
                              | to filter : [ 11 , 13 , 15 ]
                              | Piece     : 11
                              | filtered  : [ 13 ]
  [ 2   4   11  ]
                              | ----------+-------------------------------------
                              | to filter : [ 13 ]
                              | Piece     : 13
                              | filtered  : [ ]
  [ 2   4   11  13  ]

| 2 | 4 | 11 | 13 |
* * q *
q * * *
* * * q
* q * *
            1
        1
        1
    1
                              | ----------+-------------------------------------
                              | to filter : [ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8
                              |             , 9 , 10 , 11 , 12 , 13 , 14 , 15 ]
                              | Piece     : 3
                              | filtered  : [ 4 , 5 , 8 , 10 , 13 , 14 ]
  [ 3   ]
                              | ----------+-------------------------------------
                              | to filter : [ 4 , 5 , 8 , 10 , 13 , 14 ]
                              | Piece     : 4
                              | filtered  : [ 10 , 13 ]
  [ 3   4   ]
                              | ----------+-------------------------------------
                              | to filter : [ 10 , 13 ]
                              | Piece     : 10
                              | filtered  : [ ]
  [ 3   4   10  ]
            0
        0
                              | ----------+-------------------------------------
                              | to filter : [ 4 , 5 , 8 , 10 , 13 , 14 ]
                              | Piece     : 5
                              | filtered  : [ 14 ]
  [ 3   5   ]
            0
    0
2

  2
  positions have been found
  all attempts: 6
              ( 33.33% success )
  in 279ms
   ( 279 )
```

```
////////// F I R S T  Q U E E N S  O N  3 0  D I M  B O A R D ////////

| 0 | 32 | 64 | 91 | 123 | 158 | 190 | 222 | 254 | 276 | 322 | 355 | 387 | 414
| 441 | 473 | 509 | 536 | 568 | 585 | 611 | 639 | 667 | 695 | 737 | 769 | 796
| 823 | 860 | 888 |
```

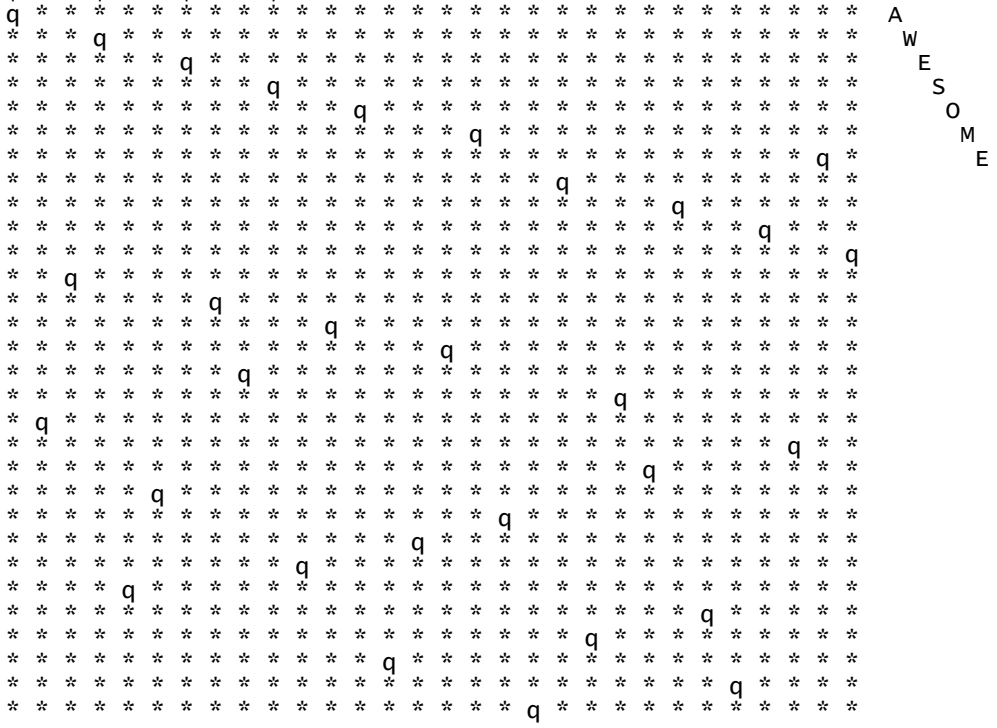


```
 1
 position has been found,
 all attempts: 20256683
              ( 0.0% success )
 in 12s 542ms
  ( 12542 )

| 0 | 33 | 66 | 99 | 121 | 154 | 187 | 220 | 253 | 290 | 324 | 357 | 379 | 413
| 446 | 455 | 509 | 535 | 554 | 598 | 602 | 646 | 668 | 711 | 731 | 768 | 795
| 822 | 862 | 887 |
```



```
 1
 position has been found,
 all attempts: 16589947
```

```
              ( 0.0% success )
  in 11s 999ms
   ( 11999 )

 | 0 | 33 | 66 | 99 | 132 | 166 | 208 | 229 | 263 | 296 | 329 | 332 | 367 | 401
 | 435 | 458 | 501 | 511 | 567 | 592 | 605 | 647 | 674 | 700 | 724 | 774 | 800
 | 823 | 865 | 888 |
```




```
  1
  position has been found,
  all attempts: 64652148
              ( 0.0% success )
  in 50s 572ms
   ( 50572 )
```

////////// THE ATTACKING MAP //////////

The visualization of the attackings.
This table represents all of the pieces by all of the pieces. The number of the bow ties in the middle is dimension minus 1. r || b == q.
"+" means the pieces located in the col and in the row can attack each other,
" " means they are not.
The command for example to show this: java -jar NqProblemExtended.jar i 5 q r f 1 n n

```
   0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
0  +  +  +  +  +  +  +           +     +        +        +     +           +
1  +  +  +  +  +  +  +  +          +     +        +        +     +
2  +  +  +  +  +     +  +  +     +     +     +        +              +
3  +  +  +  +  +        +  +  +     +     +     +        +              +
4  +  +  +  +  +           +  +        +     +     +        +  +           +
5  +  +           +  +  +  +  +  +  +           +     +        +        +
6  +  +  +        +  +  +  +  +  +  +  +           +     +        +        +
7     +  +  +     +  +  +  +  +     +  +  +     +     +     +        +
8        +  +  +  +  +  +  +  +        +  +  +     +     +     +        +
9           +  +  +  +  +  +  +           +  +        +     +     +        +
10 +     +        +  +           +  +  +  +  +  +  +           +     +
11    +     +     +  +  +        +  +  +  +  +  +  +  +           +     +
12 +     +     +     +  +  +     +  +  +  +  +     +  +  +     +     +     +
13    +     +           +  +  +  +  +  +  +  +        +  +  +     +     +
14       +     +           +  +  +  +  +  +  +           +  +        +     +
15 +        +     +     +        +  +           +  +  +  +  +  +  +
16    +        +     +     +     +  +  +        +  +  +  +  +  +  +  +
17       +        +     +     +     +  +  +     +  +  +  +  +     +  +  +
18 +        +        +     +           +  +  +  +  +  +  +  +        +  +  +
19    +        +           +     +        +  +  +  +  +  +  +           +  +
20 +           +  +        +     +     +        +  +  +           +  +  +  +  +
21    +              +        +     +     +     +  +  +           +  +  +  +  +
22       +              +        +     +     +     +  +  +     +  +  +  +  +  +
23          +     +        +        +     +           +  +  +  +  +  +  +  +
24 +           +     +        +        +     +           +  +  +  +  +  +  +
```

///// O R D E R E D A N D N O T U N I Q U E S O L U T I O N S /////

Here are some more examples of the calculations.

ordered and not unique (by mirroring) queen solutions

dimension	regular	super	awesome
1	1	1	1
2	0	0	0
3	0	0	0
4	2	0	0
5	10	0	0
6	4	0	0
7	40	0	0
8	92	0	0
9	352	0	0
10	724	4	4
11	2680	44	33
12	14200	156	6
13	73712	1876	59
14	365596	5180	8
15	2279184	32516	12

ordered and not unique (by mirroring) rook solutions

dimension	regular	super	awesome
1	1	1	1
2	2	2	2
3	6	2	1
4	24	8	8
5	120	20	10
6	720	94	22
7	5040	438	38
8	40320	2766	276
9	362880	19480	475
10	3628800	163058	2304
11	39916800	1546726	4884
12	479001600	16598282	24528

ordered and not unique (by mirroring) bishop solutions

dimension	regular	super	awesome
1	1	1	1
2	4	4	4
3	26	6	6
4	260	86	86
5	3368	854	293
6	53744	9556	2824
7	1022320	146168	12098
8	22522960	2660326	234450
9	565532992	56083228	1465563

//////// A L L A N D N O T U N I Q U E S O L U T I O N S ////////

Not unique solutions means here that the order of the placed pieces counts.

all and not unique (by mirroring) queen solutions

dimension	solutions	elapsed
1	1 (== 1 * 1!)	32ms
2	0 (== 0 * 2!)	47ms
3	0 (== 0 * 3!)	40ms
4	48 (== 2 * 4!)	47ms
5	1200 (== 10 * 5!)	31ms
6	2880 (== 4 * 6!)	62ms
7	201600 (== 40 * 7!)	125ms
8	3709440 (== 92 * 8!)	1s 884ms
9	127733760 (== 352 * 9!)	1m 3s 108ms

```
////////   F I R S T   T O   P R E P L A C E D   B I H S H O P S   ////////

Let's  calculate the positions of the chess
pieces  when they don't  attack each other!

  0 mode      (original,improved,testing) : i
  1 dimension (a positive integer)        : 8
  2 pieces    (queen,rook,bishop)         : b
  3 kinds     (regular,super,awesome)     : r
  4 hits      (ordered,all,first)         : f
  5 threads   (a positive integer)        : 1
  6 uniques   (no,yes)                    : n
  7 log       (no,info,debug)             : n
  8 placings  (ints separated by ; char)  : 43;44;45;46

Started : Sun Dec 17 15:39:23 CET 2017

Attacking map  is too  large so it
could be printed under dimension 7

 Preplaced chess pieces are: 43 44 45 46

 ( case : improved 16 )

 | 43 | 44 | 45 | 46 | 2 | 3 | 4 | 5 |
 * * b b b b * *
 * * * * * * * *
 * * * * * * * *
 * * * * * * * *
 * * * * * * * *
 * * * B B B B *
 * * * * * * * *
 * * * * * * * *

  1
  position has been found,
  in 78ms
   ( 78 )
```

Other perceptions of regular piece placings.

```
Rook position searching:
 - the success is always 100%
 - the same number of placing belongs to every first position
 - the number of valid solutions is: dimension!

Bishop position searching:
 - >70% success
 - solutions are possible from the middle or almost the end of the chessboard
 - all placings in the first or last row or column are absolutely fine solutions

Queen position searching:
 - a couple of % success
 - in case of n = 8, here are the solutions to the positions:
      4   8   16  18  18  16  8   4
      8   16  14  8   8   14  16  8
      16  14  4   12  12  4   14  16
      18  8   12  8   8   12  8   18
      18  8   12  8   8   12  8   18
      16  14  4   12  12  4   14  16
      8   16  14  8   8   14  16  8
      4   8   16  18  18  16  8   4
      - the sum of the numbers located in any row or column is:
        92 which is the final number of valid solutions
      - the number of solutions is symmetrical, not used
        ( Jk : n == 2k + 1 : we have to count the solutions until the midline,
          multiply it by 2 and add the solutions belonging to the middle element
          of the first row
          in any other case: we have to count the solutions until the midline,
          multiply it by 2 )
 - the number of placeable chess pieces can be known from the logic in every
   moment
   in case of queen:
     - before placing the 0th element: n^2 (every positions are free)
     - before placing the 1st element: n^2 - 3n + 2 (anywhere into the 1st row)
     - on the other places: it depends on the positions and the already filtered
       places
     - before placing the (n-1) element: there is exactly 1 free place.
   it is not used yet.
```