The Eight Queens Puzzle

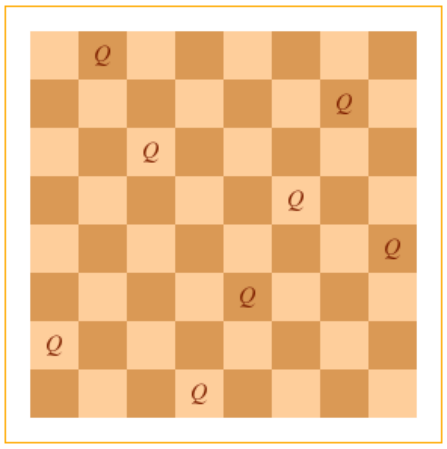
- recursive self correcting Queens solution in C

- main()
- fireworkx
- amp
- gallery
- feedback
- about

ronybc.com

n-queen.php

This program is an interesting example for recursion, taken back from the DOS era... originally coded using Turbo C. Now here is the latest Linux versions; one using ncurses to visualize the auto arrangement in action and an other faster one to reach N-Queens extreme levels like: **An arrangement of 10000 Queens.**



The famous Eight Queens problem is a chess board puzzle; to place 8 Queens on a chessboard, in such a way that none of them will be in conflict with others. That is; for every Queen sitting somewhere, there should be no other Queen in that entire row, column and the four diagonals. Trying to find such an arrangement manually is somewhat difficult. Using computers, there exists many ways to do so, better than the one decribed here as well. Here, the idea hidden deep underneath the tangled roots! of this program is to randomly place all the queens over the board and let them find their places **themselves**. This code makes them alive, intelligent and moving..! This method is a kind of brute force search, in a way, but not much. The intelligent Queens themselves resolves the problem much quicker than expected, within a few hundreds of steps, most of the time! Umm.. It cannot be said the ISO9001 way to solve the problem :) but it works very well...

Here is how it works. Place the Queens one per column, at random row. Move the Queens one at a time, vertically in her row to find a safe position where she wont face any fatty Queens in front (left side). If there is such a position pass play to the Queen behind (right). Else return play to the Queen in front (left) as 'in trouble'; that is to change the position once again to an alternate safe position. Let them run like this until they get into equilibrium..! by sheer cooperation.. thats it.. eeeh.. it is something like that..!

For example, take four nice queens A, B, C and D. And see what Queen-C does. At some point in the middle of the fight, C gets the play from B. Now C has to:

1. Get the play from B and let remaining 7 rows unvisited.
2. Check whether she is in reach of any of the queens in front (A,B)
3. If there is attack.. move to next unvisited row and check_front() again.

4. If C has tried all 8 rows and still 'moving', return play to B as 'in-Trouble'
5. Or if found a safe place, pass play to the Queen behind; queen(D)
6. If D returns 'in-Trouble', discard current and try next unvisited rows, going to step 2
7. Else if D returns 'Success', return 'Success' to B.

Queen-C has to remember which rows are visited by her until returning play back to B.

Every Queen watches only the others in her front. Here hides some priorities between the Queens that makes the foremost fatties to have to move less. That is.. most action happens at deeper levels of the recursion.

The Queens are initially placed one Queen per column at random row. This random initial row positioning of Queens chances obtaining different results between repeated executions. That way, found **92 arrangements** of 8 Queens on an 8x8 chessboard.

Original MSDOS version: Queen8.zip (obsolete)
Included C source code works with Borland Turbo-C

Linux version (LATEST):

screenshot:

```
- - - 4 - - - -
1 - - - - - - -
- - - - 5 - - -
- - - - - - - 8
- 2 - - - - - -
- - - - - - 7 -
- - 3 - - - - -
- - - - - 6 - -

number of moves = 31
conflicts = 0 0 0 0 0 0 0 0
```

Download source: n-queens-anim.c
It requires ncurses developer's libraries (libncurses5-dev) to compile 'n-queens-anim'.

```
$ apt-get install libncurses5-dev
$ gcc -O3 -o n-queens-anim n-queens-anim.c -lncurses
$ ./n-queens-anim 12
```

---=[n-queens.c]=----------

```
/*
  N ion solution for Eight Queens' Problem
  Copyright (GPL) 1999-2011 Rony B Chandran
  -----------------------------------------------------------
  website: www.ronybc.com
\
 */

#include <stdio.h>
#include <stdlib.h>
```

```
#define TROUBLE 1
#define SAFE 0

int rnd(int n)
{
        return(random() % n);
}

void print_board_stdout(int *q, int bsize)
{
        int n;
        for (n = 0; n < bsize; n++)
        {
                printf("%d, ", q[n] + 1);
        }
        printf("\n");
}

int check_front(int *q, int n)
{
        int a, b, c, x;
        a = q[n];
        b = a - n;
        c = a + n;
        for (x = 0; x < n; x++, b++, c--)
        {
                if (q[x] == a || q[x] == b || q[x] == c)
                {
                        return(TROUBLE);
                }
        }
        return(SAFE);
}

int queen(int *q, int n, int bsize, int limit)
{
        static int iteration = 0;
        int t = bsize;
        if (n > bsize - 1) return(SAFE);
        if (n == 0) iteration = 0;
        do
        {
                /* recursion */
                if (check_front(q, n) == SAFE)
                {
                        if (queen(q, n + 1, bsize, limit) == SAFE) return(
                }
                /* move_vertical */
                if (++q[n] == bsize) q[n] = 0;
        }
        /* iteration limit is optional */
        while (--t && ++iteration < limit);
        return(TROUBLE);
}

int main(int argc, char **argv)
{
        int n, bsize = 8;
        int *q;
        srandom(time(0));
        if (argc > 1) bsize = atoi(argv[1]);
        if (bsize < 1) bsize = 8;
        q = malloc(bsize * sizeof(int));
        while (1)
        {
                for (n = 0; n < bsize; n++)
                {
                        q[n] = rnd(bsize - 1);
                }

                if (queen(q, 0, bsize, bsize * bsize) == SAFE)
                {
```

```
                        print_board_stdout(q, bsize);
                }
                else /* optional; used when iteration limit enabled */
                {
                        putchar('-');
                }
        }
        return(1);
}
```

Finding all the possible arrangements

A much simpler brute force search can find all the possibilities. But here, the goal is to find out how far the above given 'self playing' algorithm will go. To make things faster, a stripped version of the original is used. With a simple comma separated list output to stdout instead the ascii drawings. Each number in the list points to the position of corresponding Queen in her row. Since it only requires single digit numbers to denote positions of eight Queens on an 8x8 chess board, the comma separation can be omitted, thus making the output an eight digit number representing the whole arrangement. For example, 42586137. Then go through following steps:

Step 1: compile...

```
$ gcc -O3 -o n-queens n-queens.c
```

Step 2: get a bucket full of output

```
$ ./n-queens |tr -d "-, " |dd bs=9 count=1000 > list-a
```

Step 3: remove all repeating entries.

```
$ sort list-a |uniq > list-b
```

Step 4: make vertical mirrored list

Eg. 47526138 -> 83162574

```
$ rev list-b > list-c
```

Step 5: combine both lists and remove identical entries.

```
$ cat list-b list-c |sort |uniq > list-d
```

Step 6: make horizontal mirrored list

Eg. 47526138 -> 52473861

```
$ cat list-d | while read Z ;do echo 99999999-$Z |bc ;done > list-e
```

Step 7: combine both lists and remove identical entries.

```
$ cat list-d list-e |sort |uniq > list-final
```

Final list of **92** possible arrangements:

15863724 - 16837425 - 17468253 - 17582463 - 24683175 - 25713864 - 25741863 - 26174835 - 26831475 - 27368514 - 27581463 - 28613574 - 31758246 - 35281746 - 35286471 - 35714286 - 35841726 - 36258174 - 36271485 - 36275184 - 36418572 - 36428571 - 36814752 - 36815724 - 36824175 - 37285146 - 37286415 - 38471625 - 41582736 - 41586372 - 42586137 - 42736815 - 42736851 - 42751863 - 42857136 - 42861357 - 46152837 - 46827135 - 46831752 - 47185263 - 47382516 - 47526138 - 47531682 - 48136275 - 48157263 - 48531726 - 51468273 - 51842736 - 51863724 - 52468317 - 52473861 - 52617483 - 52814736 - 53168247 - 53172864 - 53847162 -

57138642 - 57142863 - 57248136 - 57263148 - 57263184 - 57413862 - 58413627 - 58417263 - 61528374 - 62713584 - 62714853 - 63175824 - 63184275 - 63185247 - 63571428 - 63581427 - 63724815 - 63728514 - 63741825 - 64158273 - 64285713 - 64713528 - 64718253 - 68241753 - 71386425 - 72418536 - 72631485 - 73168524 - 73825164 - 74258136 - 74286135 - 75316824 - 82417536 - 82531746 - 83162574 - 84136275

An arrangement of 10000 Queens:

```
$ ./n-queens 10000
```

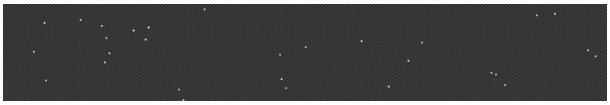
This attempt took 20 hours and 40 minutes to finish. The machine used is an AMD Athlon II X2 240 Processor with 1GB DDR2 RAM, running 64-bit Ubuntu 10.04. The time taken cannot be considered exact because the machine was not left unused for being dedicated to the task. And this method **relies on random.** It takes a random distribution of queens and tries to correct it with a limited (optional) number of changes; else tries next random. The following result came out after 122 retries. ie. 99.18% of the process were fully discarded.

n-queen-10000.list : original output.
n-queen-10000.png : PNG image, 10000 x 10000 pixels.
NOTE: very big image, requires lots of memory.
Direct following the link may crash the browser; do download and open in a good image viewer.

In order to view it fast and to zoom appropriately without getting the squares blurred, use nearest neighbor interpolation in your image viewer by turning off smooth or high quality zooming. White dots (pixels) are Queens and two dark shades for pixels representing empty squares. The board is so big that rendering just a single pixel for each tile resulted in a mammoth image weighing 100 Megapixels. If converted to standard chessboard dimensions, this one will have more than half kilometers length and width. Without zooming (1:1) the white dots appears arranged like stars in a clear night sky. Full of stars, like when far away from streetlight pollution. None of them aligned to any, scattered over vast voids. And most interestingly there are nice shaped constellations too. A beautiful sight as a pay for going after ten thousand Queens.

N Queens - 1000:

```
$ ./n-queens 1000
```



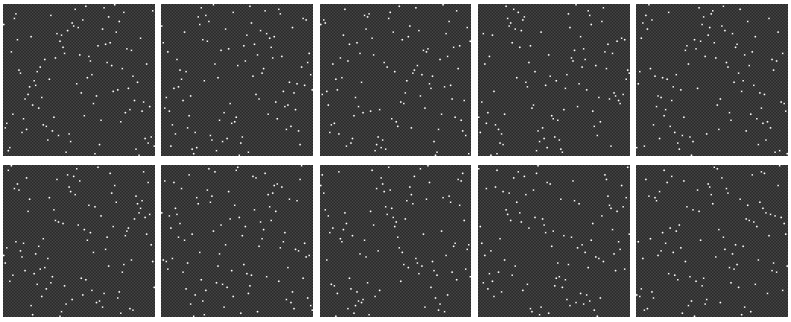
In numbers:
906, 898, 280, 387, 701, 247, 647, 696, 596, 648, 694, 644, 491, 478, 307, 877, 884, 598, 60, 109, 135, 886, 608, 486, 229, 390, 434, 986, 167, 505, 937, 74, 124, 217, 178, 823, 463, 543, 238, 777, 191, 651, 140, 400, 847, 165, 277, 730, 481, 55, 557, 615, 658, 883, 820, 604, 273, 254, 591, 439, 477, 527, 230, 599, 462, 126, 141, 925, 668, 98, 422, 579,

467, 561, 697, 314, 444, 692, 763, 926, 465, 321, 259, 128, 922, 79, 728, 196, 51, 318, 354, 528, 564, 301, 845, 26, 427, 705, 669, 813, 802, 91, 111, 987, 370, 806, 302, 814, 218, 784, 457, 681, 821, 716, 804, 744, 794, 251, 656, 565, 568, 9, 92, 850, 310, 936, 876, 455, 642, 264, 989, 162, 355, 97, 150, 442, 621, 169, 974, 837, 952, 431, 518, 492, 865, 42, 233, 380, 293, 889, 943, 580, 897, 35, 429, 927, 688, 24, 99, 49, 287, 86, 929, 361, 900, 796, 803, 243, 964, 776, 78, 633, 928, 313, 125, 791, 356, 358, 888, 646, 963, 829, 942, 582, 584, 372, 507, 988, 114, 325, 37, 401, 127, 683, 759, 745, 479, 279, 984, 164, 772, 779, 514, 698, 93, 637, 207, 166, 711, 94, 529, 675, 923, 470, 255, 504, 559, 480, 494, 671, 520, 249, 789, 645, 649, 269, 391, 129, 544, 376, 10, 317, 153, 525, 15, 965, 159, 940, 131, 589, 33, 655, 263, 955, 853, 236, 179, 404, 433, 392, 793, 951, 357, 583, 600, 3, 848, 706, 849, 405, 80, 857, 428, 950, 100, 161, 632, 975, 102, 760, 567, 130, 134, 545, 805, 979, 500, 700, 103, 930, 807, 892, 887, 880, 475, 198, 881, 40, 902, 449, 155, 702, 25, 577, 652, 122, 454, 284, 101, 550, 768, 378, 398, 613, 921, 924, 316, 418, 619, 408, 68, 430, 18, 949, 23, 210, 147, 904, 248, 53, 72, 399, 471, 815, 691, 121, 938, 145, 120, 755, 413, 601, 851, 811, 212, 485, 733, 241, 624, 70, 654, 689, 213, 665, 638, 239, 871, 503, 142, 838, 270, 931, 953, 741, 747, 657, 574, 403, 511, 414, 870, 920, 12, 714, 734, 941, 199, 182, 184, 819, 253, 546, 509, 461, 934, 868, 699, 531, 88, 549, 362, 353, 482, 28, 808, 223, 673, 394, 338, 901, 513, 204, 822, 244, 917, 268, 193, 115, 448, 83, 653, 416, 346, 885, 873, 990, 460, 571, 512, 547, 118, 585, 616, 311, 606, 143, 256, 994, 250, 588, 614, 757, 797, 437, 997, 720, 419, 907, 540, 590, 703, 912, 993, 47, 788, 587, 39, 966, 152, 267, 231, 985, 843, 846, 300, 168, 704, 548, 163, 945, 852, 775, 707, 636, 948, 417, 61, 50, 322, 319, 639, 16, 232, 340, 66, 736, 939, 826, 712, 798, 81, 946, 498, 640, 497, 795, 810, 911, 58, 685, 858, 622, 176, 278, 261, 89, 693, 34, 859, 6, 365, 208, 21, 298, 260, 800, 44, 185, 623, 451, 715, 423, 108, 195, 64, 305, 717, 862, 219, 488, 272, 75, 104, 441, 65, 350, 246, 752, 106, 110, 756, 453, 304, 501, 749, 560, 17, 508, 473, 351, 958, 891, 769, 52, 84, 553, 359, 780, 132, 288, 265, 393, 363, 382, 832, 144, 438, 85, 890, 522, 894, 375, 723, 909, 860, 466, 190, 874, 944, 385, 947, 609, 245, 735, 667, 320, 971, 729, 816, 96, 27, 73, 490, 105, 148, 31, 225, 290, 828, 833, 516, 435, 201, 211, 329, 778, 369, 521, 377, 32, 893, 323, 641, 136, 742, 4, 183, 713, 447, 976, 809, 186, 790, 11, 983, 627, 761, 932, 916, 576, 773, 420, 13, 678, 371, 46, 458, 446, 286, 831, 450, 882, 174, 801, 718, 905, 825, 879, 611, 967, 569, 116, 154, 384, 175, 854, 962, 896, 770, 595, 203, 274, 722, 899, 935, 69, 959, 386, 214, 222, 206, 662, 112, 48, 468, 844, 933, 992, 396, 228, 663, 960, 56, 526, 294, 188, 373, 999, 781, 872, 271, 670, 113, 991, 573, 738, 782, 252, 117, 1, 436, 30, 364, 240, 786, 827, 754, 424, 532, 151, 650, 187, 908, 726, 721, 915, 603, 818, 812, 133, 661, 87, 489, 493, 783, 95, 216, 283, 275, 43, 281, 409, 54, 659, 366, 839, 181, 146, 257, 395, 980, 630, 570, 830, 82, 5, 664, 368, 496, 180, 205, 861, 954, 660, 324, 737, 710, 227, 59, 679, 258, 8, 835, 20, 328, 192, 554, 506, 995, 523, 878, 708, 197, 189, 495, 914, 957, 875, 972, 381, 731, 177, 996, 680, 817, 262, 123, 220, 484, 864, 607, 456, 556, 383, 464, 970, 276, 22, 170, 961, 534, 682, 666, 410, 562, 866, 29, 459, 452, 19, 517, 156, 139, 472, 538, 674, 440, 379, 541, 686, 973, 224, 799, 515, 237, 297, 137, 235, 998, 2, 149, 295, 575, 869, 339, 171, 677, 345, 291, 138, 57, 524, 62, 895, 690, 487, 530, 840, 592, 226, 499, 551, 903, 282, 41, 824, 303, 842, 14, 956, 740, 158, 172, 306, 977, 510, 157, 299, 594, 202, 173, 620, 695, 107, 535, 221, 296, 725, 910, 834, 502, 266, 388, 349, 209, 200, 160, 336, 242, 867, 285, 724, 7, 292, 719, 913, 407, 578, 969, 289, 750, 71, 312, 234, 787, 919, 330, 194, 309, 836, 856, 732, 981, 978, 751, 968, 308, 626, 90, 119, 536, 327, 483, 542, 331, 76, 352, 558, 389, 347, 215, 333, 762, 539, 918, 552, 982, 334, 343, 326, 374, 432, 597, 77, 335, 397, 555, 341, 727, 537, 586, 402, 469, 593, 332, 415, 412, 443, 563, 746, 315, 739, 566, 753, 45, 612, 425, 602, 406, 519, 581, 445, 426, 421, 367, 348, 360, 572, 605, 474, 764,

533, 610, 1000, 743, 618, 625, 841, 774, 635, 643, 344, 748, 63, 628, 855, 672, 617, 476, 684, 792, 67, 337, 771, 785, 631, 758, 863, 342, 676, 709, 634, 767, 411, 687, 766, 36, 629, 765, 38.

N Queens - 100:

```
$ ./n-queens 100
```



Not diggin further than ten thousand queens, because it's just useless. BTW... praising my mighty μP, The AMD Athlon II X2 2.8GHz.