Solving the n-Queens Problem using Local Search

Instructions

Total Points: 10

Complete this notebook and submit it. The notebook needs to be a complete project report with

- **your implementation,**
- **documentation including a short discussion of how your implementation works and your design choices, and**
- **experimental results (e.g., tables and charts with simulation results) with a short discussion of what they mean.**

Use the provided notebook cells and insert additional code and markdown cells as needed.

The n-Queens Problem

- **Goal: Find an arrangement of** n **queens on a** $n \times n$ **chess board so that no queen is on the same row, column or diagonal as any other queen.**
- **State space: An arrangement of the queens on the board. We restrict the state space to arrangements where there is only a single queen per column. We represent a state as an integer vector** $\mathbf{q} = \{q_1, q_2, \ldots, q_n\}$ **, each number representing the row positions of the queens from left to right. We will call a state a "board."**
- **Objective function: The number of pairwise conflicts (i.e., two queens in the same row/column/diagonal). The optimization problem is to find the optimal arrangement** $\mathbf{q}^*$ **of** n **queens on the board can be written as:**

  > **minimize:** $\text{conflicts}(\mathbf{q})$
  >
  > **subject to:** $\mathbf{q}$ contains only one queen per column

  Note: the constraint (subject to) is enforced by the definition of the state space.
- **Local improvement move: Move one queen to a different row in its column.**
- **Termination: For this problem there is always an arrangement** $\mathbf{q}^*$ **with** $\text{conflicts}(\mathbf{q}^*) = 0$ **, however, the local improvement moves might end up in a local minimum.**

Helper functions

In [54]:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors

np.random.seed(1234)


def random_board(n):
    """Creates a random board of size n x n. Note that only a single queen is placed in
each column!"""

    return(np.random.randint(0,n, size = n))

def comb2(n): return n*(n-1)//2 # this is n choose 2 equivalent to math.comb(n, 2); // i
s int division
```

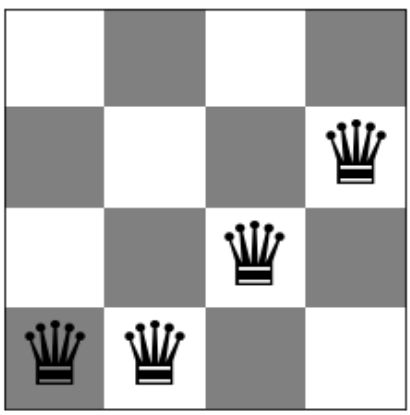
```
def conflicts(board):
    """Caclulate the number of conflicts, i.e., the objective function."""

    n = len(board)

    horizontal_cnt = [0] * n
    diagonal1_cnt = [0] * 2 * n
    diagonal2_cnt = [0] * 2 * n

    for i in range(n):
        horizontal_cnt[board[i]] += 1
        diagonal1_cnt[i + board[i]] += 1
        diagonal2_cnt[i - board[i] + n] += 1

    return sum(map(comb2, horizontal_cnt + diagonal1_cnt + diagonal2_cnt))


def show_board(board, cols = ['white', 'gray'], fontsize = 48):
    """display the board"""

    n = len(board)

    # create chess board display
    display = np.zeros([n,n])
    for i in range(n):
        for j in range(n):
            if (((i+j) % 2) != 0):
                display[i,j] = 1

    cmap = colors.ListedColormap(cols)
    fig, ax = plt.subplots()
    ax.imshow(display, cmap = cmap,
              norm = colors.BoundaryNorm(range(len(cols)+1), cmap.N))
    ax.set_xticks([])
    ax.set_yticks([])

    # place queens. Note: Unicode u265B is a black queen
    for j in range(n):
        plt.text(j, board[j], u"\u265B", fontsize = fontsize,
                 horizontalalignment = 'center',
                 verticalalignment = 'center')

    print(f"Board with {conflicts(board)} conflicts.")
    plt.show()
```

Create a board

In [55]:

```
board = random_board(4)

show_board(board)
print(f"Queens (left to right) are at rows: {board}")
print(f"Number of conflicts: {conflicts(board)}")
```

```
Board with 4 conflicts.
```



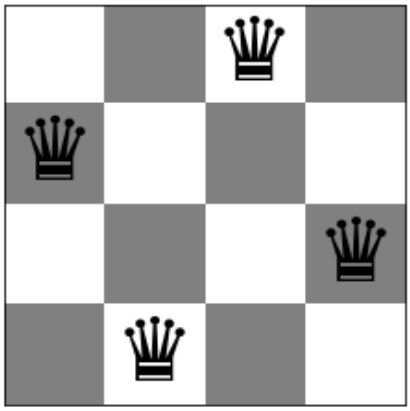
```
Queens (left to right) are at rows: [3 3 2 1]
Number of conflicts: 4
```

A board 4×4
with no conflicts:

In [56]:

```
board = [1,3,0,2]
show_board(board)
```

```
Board with 0 conflicts.
```



Steepest-ascend Hill Climbing Search [2 Points]

Calculate the objective function for all local moves (move each queen within its column) and always choose the best among all local moves. If there are no local moves that improve the objective, then you have reached a local optimum.

In [57]:

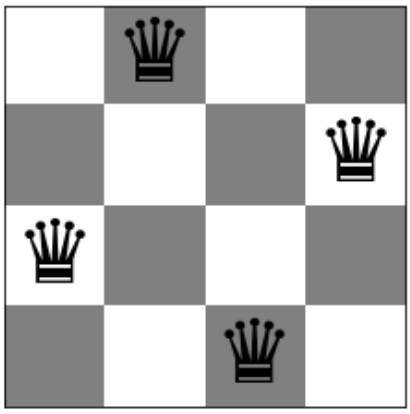
```
def get_poss_moves(board, n):
    pos = []
    for i in range(n):
        pos.append(i)
    temp = pos.copy()
    moves = []
    for i in range(n):
        m = [j for j in pos if j != board[i]]
        moves.append(m)

    return moves

def get_uphill(board, n, moves):
    conflict = conflicts(board)
    uphill = []
    for i, m in enumerate(moves):
        temp = board.copy()
        for j in m:
            temp[i] = j
            if conflicts(temp) < conflict:
                uphill = [i, j]
                conflict = conflicts(temp)

    return uphill

def steep_ascend(n, moves):
    board = random_board(n)

    if conflicts(board) == 0:
        return 0, 0, board, False

    cnt = iters = 0
```

```
    while conflicts(board) != 0:
        poss_moves = get_poss_moves(board, n)
        uphill_move = get_uphill(board, n, poss_moves)

        iters += 1
        if len(uphill_move) != 0:
            board[uphill_move[0]] = uphill_move[1]
            cnt = 0
        else:
            cnt += 1

        if cnt > moves:
            return iters, conflicts(board), board, True

    return iters, conflicts(board), board, False
```

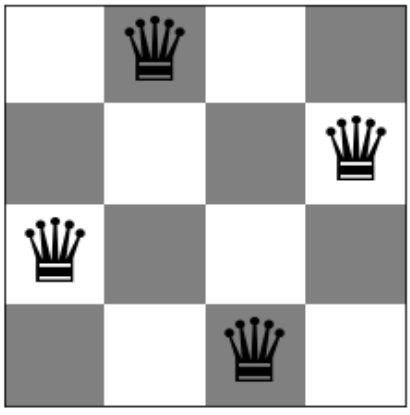
In [58]:

```
for i in range(4):
    output = steep_ascend(4, 100)
    print("Board #" + str(i+1))
    if output[3]:
        print("Stuck at local max.")
    else:
        print("Solved")
    print("Conflicts: " + str(output[1]))
    print("Iterations: "+ str(output[0]))
    show_board(output[2])
```

```
Board #1
Solved
Conflicts: 0
Iterations: 2
Board with 0 conflicts.
```

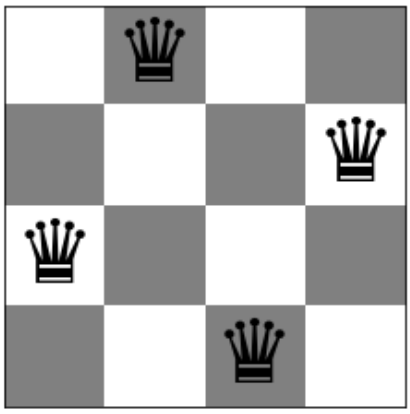


```
Board #2
Solved
Conflicts: 0
Iterations: 2
Board with 0 conflicts.
```

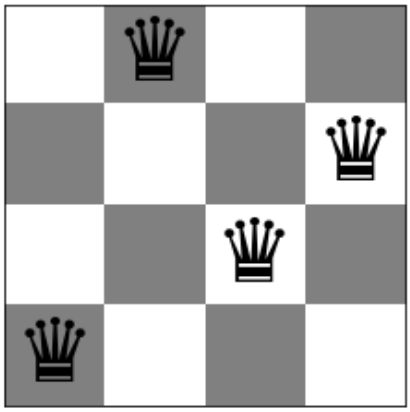


```
Board #3
Solved
```

```
Conflicts: 0
Iterations: 2
Board with 0 conflicts.
```



```
Board #4
Stuck at local max.
Conflicts: 1
Iterations: 103
Board with 1 conflicts.
```



Stochastic Hill Climbing 1 [1 Point]

Chooses randomly from among all uphill moves. Make the probability of the choice proportional to the steepness of the uphill move (i.e., with the improvement in conflicts).

In [59]:

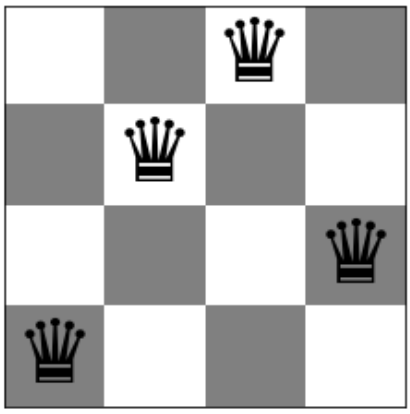
```
import random

def stochastic1(n, moves):
    board = random_board(n)

    if conflicts(board) == 0:
        return 0, 0, board, False

    cnt = iters = 0
    while conflicts(board) != 0:
        poss_moves = get_poss_moves(board, n)
        uphill_moves = get_uphill(board, n, poss_moves)

        iters += 1
        if len(uphill_moves) != 0:
            random.shuffle(uphill_moves)
            board[uphill_moves[0]] = uphill_moves[1]
            cnt = 0
        else:
            cnt += 1

        if cnt > moves:
            return iters, conflicts(board), board, True
```

```
    return iters, conflicts(board), board, False
```

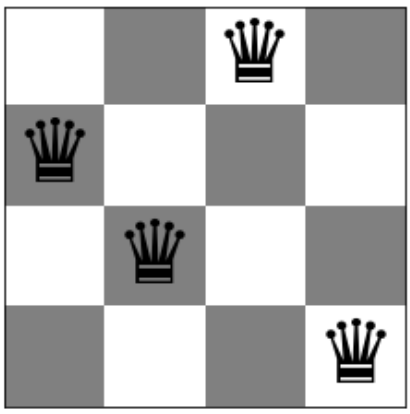
In [60]:

```
for i in range(4):
    output = stochastic1(4, 100)
    print("Board #" + str(i+1))
    if output[3]:
        print("Stuck at local max.")
    else:
        print("Solved")
    print("Conflicts: " + str(output[1]))
    print("Iterations: " + str(output[0]))
    show_board(output[2])
```

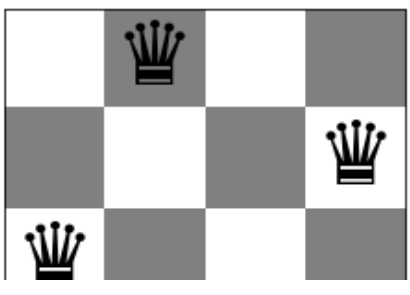
```
Board #1
Stuck at local max.
Conflicts: 1
Iterations: 106
Board with 1 conflicts.
```



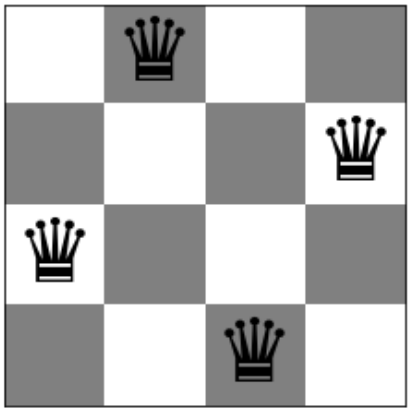
```
Board #2
Stuck at local max.
Conflicts: 1
Iterations: 103
Board with 1 conflicts.
```



```
Board #3
Solved
Conflicts: 0
Iterations: 2
Board with 0 conflicts.
```



```
Board #4
Solved
Conflicts: 0
Iterations: 5
Board with 0 conflicts.
```



Stochastic Hill Climbing 2 [2 Point]

A popular version of stochastic hill climbing generates only a single random local neighbor at a time and accept it if it has a better objective function value than the current state. This is very efficient if each state has many possible successor states. This method is called "First-choice hill climbing" in the textbook.

Notes:

- **Detecting local optima is tricky! You can, for example, stop if you were not able to improve the objective function during the last x tries.**

In [61]:

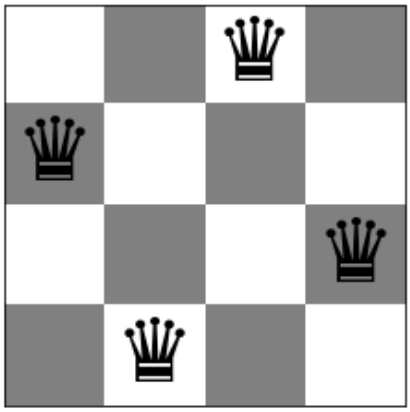
```
def find_neighbor(board, n, iters):
    neighbor = []
    rand_i = random.randint(0, n - 1)
    conflict = conflicts(board)
    temp = board.copy()
    cnt = 0

    while len(neighbor) <= 0:
        for i in range(n):
            if board[rand_i] != i:
                temp[rand_i] = i
                if conflicts(temp) < conflict:
                    neighbor = [rand_i, i]
        rand_i = random.randint(0, n - 1)
        cnt += 1
        if cnt > iters:
            return neighbor

    return neighbor

def stochastic2(n, moves):
    board = random_board(n)

    if conflicts(board) == 0:
        return 0, 0, board, False

    cnt = iters = 0
    while conflicts(board) != 0:
```

```
        neighbor = find_neighbor(board, n, moves)
        iters += 1

        if len(neighbor) != 0:
            board[neighbor[0]] = neighbor[1]
            cnt = 0
        else:
            cnt += 1

        if cnt > moves:
            return iters, conflicts(board), board, True

    return iters, conflicts(board), board, False
```

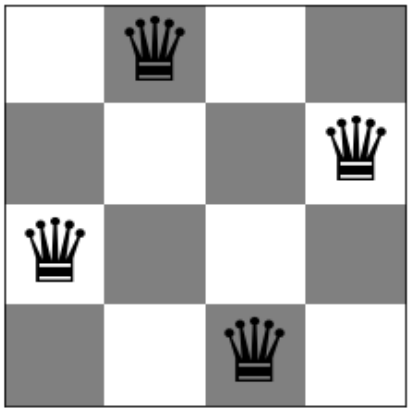
In [62]:

```
for i in range(4):
    output = stochastic2(4, 100)
    print("Board #" + str(i+1))
    if output[3]:
        print("Stuck at local max.")
    else:
        print("Solved")
    print("Conflicts: " + str(output[1]))
    print("Iterations: "+ str(output[0]))
    show_board(output[2])
```

```
Board #1
Solved
Conflicts: 0
Iterations: 59
Board with 0 conflicts.
```

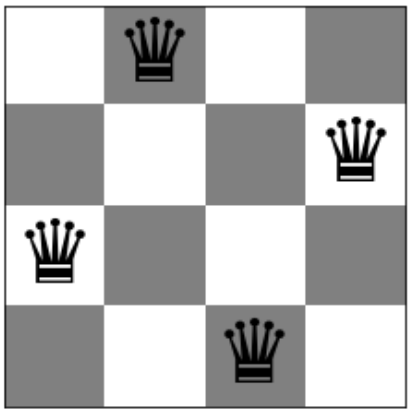


```
Board #2
Solved
Conflicts: 0
Iterations: 6
Board with 0 conflicts.
```

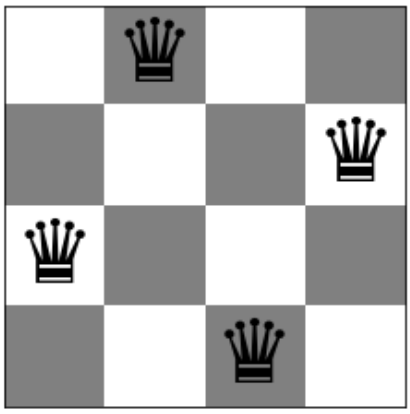


```
Board #3
Solved
Conflicts: 0
Iterations: 17
```

```
Board with 0 conflicts.
```



```
Board #4
Solved
Conflicts: 0
Iterations: 9
Board with 0 conflicts.
```



Hill Climbing Search with Random Restarts [1 Point]

Hill climbing will often end up in local optima. Restart the each of the three hill climbing algorithm up to 100 times with a random board to find a better (hopefully optimal) solution.

In [63]:

```
def random_steep(n, moves, restarts):
    board = random_board(n)

    if conflicts(board) == 0:
        return 0, 0, board, False

    cnt = iters = 0
    while conflicts(board) != 0:
        poss_moves = get_poss_moves(board, n)
        uphill_move = get_uphill(board, n, poss_moves)

        iters += 1
        if len(uphill_move) != 0:
            board[uphill_move[0]] = uphill_move[1]
            cnt = 0
        else:
            cnt += 1

        if cnt > moves:
            board = random_board(n)
            cnt = 0
            restarts -= 1
        if restarts == 0:
            return iters, conflicts(board), board, True
```

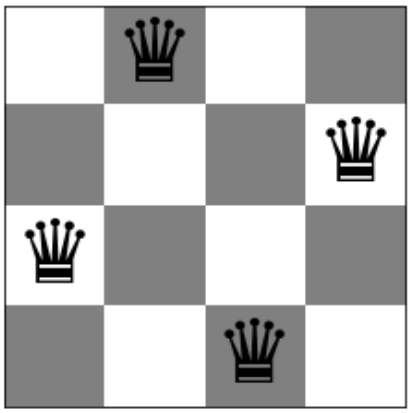
```
    return iters, conflicts(board), board, False

def random_sto1(n, moves, restarts):
    board = random_board(n)

    if conflicts(board) == 0:
        return 0, 0, board, False

    cnt = iters = 0
    while conflicts(board) != 0:
        poss_moves = get_poss_moves(board, n)
        uphill_moves = get_uphill(board, n, poss_moves)

        iters += 1
        if len(uphill_moves) != 0:
            random.shuffle(uphill_moves)
            board[uphill_moves[0]] = uphill_moves[1]
            cnt = 0
        else:
            cnt += 1

        if cnt > moves:
            board = random_board(n)
            cnt = 0
            restarts -= 1
        if restarts == 0:
            return iters, conflicts(board), board, True

    return iters, conflicts(board), board, False

def random_sto2(n, moves, restarts):
    board = random_board(n)

    if conflicts(board) == 0:
        return 0, 0, board, False

    cnt = iters = 0
    while conflicts(board) != 0:
        neighbor = find_neighbor(board, n, moves)
        iters += 1

        if len(neighbor) != 0:
            board[neighbor[0]] = neighbor[1]
            cnt = 0
        else:
            cnt += 1

        if cnt > moves:
            board = random_board(n)
            cnt = 0
            restarts -= 1
        if restarts == 0:
            return iters, conflicts(board), board, True

    return iters, conflicts(board), board, False
```

In [64]:

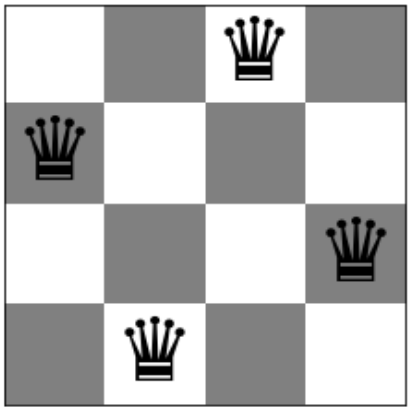
```
algos = ["Steepest Ascend Hill-Climbing", "Stochastic Hill Climbing 1", "Stochastic Hill
Climbing 2"]

for i in range(len(algos)):
    if i == 0:
        output = random_steep(4, 100, 100)
    if i == 1:
        output = random_sto1(4, 100, 100)
    else:
        output = random_sto2(4, 100, 100)

    print(algos[i])
    if output[3]:
```

```
        print("Stuck at local max.")
    else:
        print("Solved")
    print("Conflicts: " + str(output[1]))
    print("Iterations: "+ str(output[0]))
    show_board(output[2])
```

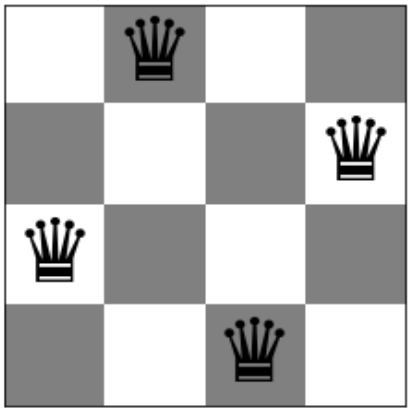
```
Steepest Ascend Hill-Climbing
Solved
Conflicts: 0
Iterations: 322
Board with 0 conflicts.
```



```
Stochastic Hill Climbing 1
Solved
Conflicts: 0
Iterations: 5
Board with 0 conflicts.
```



```
Stochastic Hill Climbing 2
Solved
Conflicts: 0
Iterations: 16
Board with 0 conflicts.
```



Simulated Annealing [2 Points]

Simulated annealing is a form of stochastic hill climbing that also allows downhill moves with a probability proportional to the temperature. This is done to avoid local optima. The temperature is decreased in every iteration following an annealing schedule. You have to experiment with the annealing schedule (Google to find guidance on this).

In [65]:

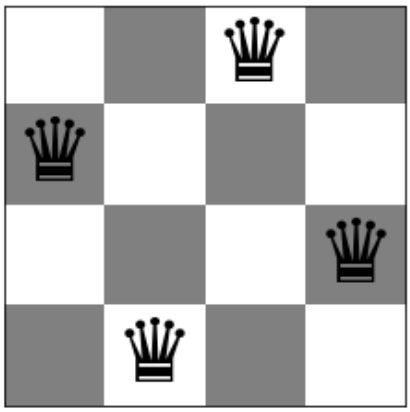
```
def find_moves(board, n, maxIters, temp):
    moves = []
    rand_i = random.randint(0, n - 1)
    conflict = conflicts(board)
    temp_board = board.copy()
    cnt = 0

    while len(moves) <= 0:
        for i in range(n):
            if board[rand_i] != i:
                temp_board[rand_i] = i
                if conflicts(temp_board) < conflict:
                    moves = [rand_i, i]
                else:
                    ran = random.random()
                    if ran < temp:
                        moves = [rand_i, i]

        rand_i = random.randint(0, n - 1)
        cnt += 1
        if cnt > maxIters:
            return moves

    return moves

def sim_annealing(n, maxMoves, maxIters):
    board = random_board(n)

    if conflicts(board) == 0:
        return 0, 0, board, False

    cnt = iters = 0
    while conflicts(board) != 0:
        iters += 1
        temperature = 50 * (.8 ** iters)
        moves = find_moves(board, n, maxIters, temperature)

        if len(moves) != 0:
            board[moves[0]] = moves[1]
            cnt = 0
        else:
            cnt += 1

        if cnt > maxMoves or iters > maxIters:
            return iters, conflicts(board), board, True

    return iters, conflicts(board), board, False
```

In [66]:

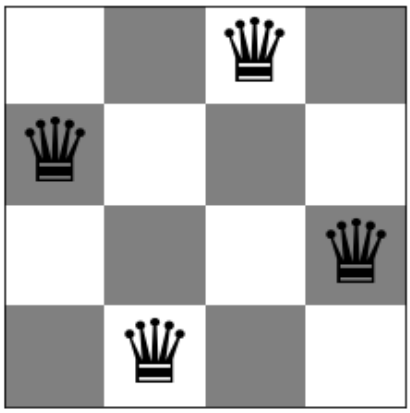
```
for i in range(4):
    output = sim_annealing(4, 100, 500)
    print("Board #" + str(i+1))
    if output[3]:
        print("Stuck at local max.")
    else:
        print("Solved")
    print("Conflicts: " + str(output[1]))
    print("Iterations: "+ str(output[0]))
    show_board(output[2])
```

```
Board #1
Solved
Conflicts: 0
```

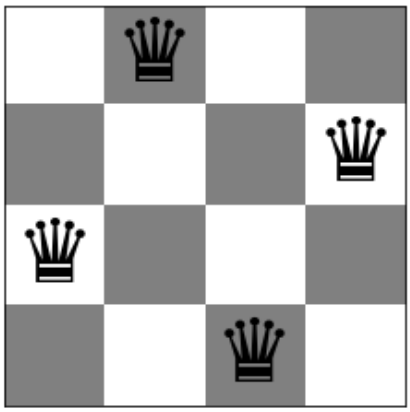
Iterations: 34
Board with 0 conflicts.



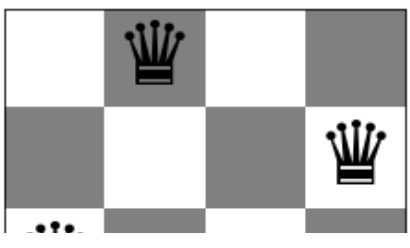
Board #2
Solved
Conflicts: 0
Iterations: 42
Board with 0 conflicts.



Board #3
Solved
Conflicts: 0
Iterations: 23
Board with 0 conflicts.



Board #4
Solved
Conflicts: 0
Iterations: 27
Board with 0 conflicts.



Compare Performance [2 Points]

Use runtime and objective function value to compare the algorithms.

- **Use boards of different sizes to explore how the different algorithms perform. Make sure that you run the algorithms for each board size several times (at least 10 times) with different starting boards and report averages.**
- **How do the algorithms scale with problem size?**
- **What is the largest board each algorithm can solve in a reasonable amount time?**

The example below times creating 100 random boards and calculating the conflicts. Reported is the average run time over `N = 100` runs.

For timing you can use the `time` package.

In [67]:

```
import time

N = 100
total = 0

for i in range(N):
    t0 = time.time()
    for i in range(1,100): conflicts(random_board(8))
    t1 = time.time()
    total += t1 - t0

tm = total/N

print(f"This took: {tm * 1e3} milliseconds")
```

```
This took: 4.172546863555908 milliseconds
```

The `timit` package is useful to measure time for code that is called repeatedly.

In [68]:

```
import timeit

N = 100

tm = timeit.timeit('for i in range(1,100): conflicts(random_board(8))',
                   globals = globals(), number = N)/N

print(f"This took: {tm * 1e3} milliseconds")
```

```
This took: 4.494981999996526 milliseconds
```

In [82]:

```
import time

N = 10
algos = ["Steepest Ascend", "Stochastic 1", "Stochastic 2", "Simulated Annealing",
         "Steepest Ascend w/ Restarts", "Stochastic 1 w/ Restarts", "Stochastic 2 w/ Res
tarts"]
sizes = [4, 5, 6, 7, 8]
```

```
for size in range(len(sizes)):
    print("Board Size: " + str(sizes[size]))
    print("--------------")
    for algo in range(len(algos)):
        iters = cons = times = 0
        for i in range(N):
            start = time.time()
            if algo == 0:
                output = steep_ascend(sizes[size], 1000)
            if algo == 1:
                output = stochastic1(sizes[size], 1000)
            if algo == 2:
                output = stochastic2(sizes[size], 1000)
            if algo == 3:
                output = sim_annealing(sizes[size], 1000, 5000)
            if algo == 4:
                output = random_steep(sizes[size], 1000, 1000)
            if algo == 5:
                output = random_sto1(sizes[size], 1000, 1000)
            else:
                output = random_sto2(sizes[size], 1000, 1000)
            end = time.time()

            times += end - start
            iters += output[0]
            cons += output[1]

        print("Algorithm: " + algos[algo])
        print("Average Time: " + str(times / N))
        print("Average Iterations: " + str(iters / N))
        print("Average Conflicts: " + str(cons / N))
        print()
```

```
Board Size: 4
--------------
Algorithm: Steepest Ascend
Average Time: 0.2782069444656372
Average Iterations: 59.8
Average Conflicts: 0.0

Algorithm: Stochastic 1
Average Time: 0.125730562210083
Average Iterations: 27.0
Average Conflicts: 0.0

Algorithm: Stochastic 2
Average Time: 0.26781015396118163
Average Iterations: 26.4
Average Conflicts: 0.0

Algorithm: Simulated Annealing
Average Time: 0.41320195198059084
Average Iterations: 32.5
Average Conflicts: 0.0

Algorithm: Steepest Ascend w/ Restarts
Average Time: 0.09493005275726318
Average Iterations: 27.7
Average Conflicts: 0.0

Algorithm: Stochastic 1 w/ Restarts
Average Time: 0.03326649665832519
Average Iterations: 242.7
Average Conflicts: 0.0

Algorithm: Stochastic 2 w/ Restarts
Average Time: 0.10956473350524902
Average Iterations: 42.7
Average Conflicts: 0.0

Board Size: 5
--------------
```

```
Algorithm: Steepest Ascend
Average Time: 0.056219863891601565
Average Iterations: 14.9
Average Conflicts: 0.0

Algorithm: Stochastic 1
Average Time: 0.05231554508209228
Average Iterations: 13.1
Average Conflicts: 0.0

Algorithm: Stochastic 2
Average Time: 0.039409947395324704
Average Iterations: 7.7
Average Conflicts: 0.0

Algorithm: Simulated Annealing
Average Time: 0.25234060287475585
Average Iterations: 14.0
Average Conflicts: 0.0

Algorithm: Steepest Ascend w/ Restarts
Average Time: 0.08406119346618653
Average Iterations: 20.8
Average Conflicts: 0.0

Algorithm: Stochastic 1 w/ Restarts
Average Time: 0.018891286849975587
Average Iterations: 79.4
Average Conflicts: 0.0

Algorithm: Stochastic 2 w/ Restarts
Average Time: 0.03314368724822998
Average Iterations: 11.7
Average Conflicts: 0.0

Board Size: 6
--------------
Algorithm: Steepest Ascend
Average Time: 3.5157685041427613
Average Iterations: 569.4
Average Conflicts: 0.0

Algorithm: Stochastic 1
Average Time: 4.3700706481933596
Average Iterations: 709.6
Average Conflicts: 0.0

Algorithm: Stochastic 2
Average Time: 3.8653354406356812
Average Iterations: 529.6
Average Conflicts: 0.0

Algorithm: Simulated Annealing
Average Time: 7.3768822193145756
Average Iterations: 614.9
Average Conflicts: 0.0

Algorithm: Steepest Ascend w/ Restarts
Average Time: 4.995322561264038
Average Iterations: 731.6
Average Conflicts: 0.0

Algorithm: Stochastic 1 w/ Restarts
Average Time: 0.793758749961853
Average Iterations: 1938.0
Average Conflicts: 0.0

Algorithm: Stochastic 2 w/ Restarts
Average Time: 3.7451581001281737
Average Iterations: 610.1
Average Conflicts: 0.0
```

```
Board Size: 7
--------------
Algorithm: Steepest Ascend
Average Time: 3.148687505722046
Average Iterations: 373.5
Average Conflicts: 0.0

Algorithm: Stochastic 1
Average Time: 1.4682157993316651
Average Iterations: 174.5
Average Conflicts: 0.0

Algorithm: Stochastic 2
Average Time: 2.084240508079529
Average Iterations: 180.9
Average Conflicts: 0.0

Algorithm: Simulated Annealing
Average Time: 3.4515942096710206
Average Iterations: 118.6
Average Conflicts: 0.0

Algorithm: Steepest Ascend w/ Restarts
Average Time: 1.2870989561080932
Average Iterations: 134.2
Average Conflicts: 0.0

Algorithm: Stochastic 1 w/ Restarts
Average Time: 0.29818804264068605
Average Iterations: 467.5
Average Conflicts: 0.0

Algorithm: Stochastic 2 w/ Restarts
Average Time: 1.468738579750061
Average Iterations: 182.5
Average Conflicts: 0.0

Board Size: 8
--------------
Algorithm: Steepest Ascend
Average Time: 4.294652533531189
Average Iterations: 393.2
Average Conflicts: 0.0

Algorithm: Stochastic 1
Average Time: 4.2119303941726685
Average Iterations: 387.2
Average Conflicts: 0.0

Algorithm: Stochastic 2
Average Time: 4.5176966190338135
Average Iterations: 334.6
Average Conflicts: 0.0

Algorithm: Simulated Annealing
Average Time: 9.728964352607727
Average Iterations: 550.4
Average Conflicts: 0.0

Algorithm: Steepest Ascend w/ Restarts
Average Time: 6.46439254283905
Average Iterations: 551.5
Average Conflicts: 0.0

Algorithm: Stochastic 1 w/ Restarts
Average Time: 0.8494309902191162
Average Iterations: 895.1
Average Conflicts: 0.0

Algorithm: Stochastic 2 w/ Restarts
Average Time: 4.52832338809967
Average Iterations: 419.5
```

```
Average Conflicts: 0.0
```

Discussion

It was very odd to see that the runs I did with a board size of 5, all the algorithms ran faster with less iterations than the board of size 4. However, there was a huge jump in runtime and average iterations from size 5 to size 6. I am wondering if this is an issue with the way my machine is running because these results are totally counter-intuitive. While the algorithms ran in appropriate time with sizes 4 and 5, they began to take whole seconds to run through the board of size 6. This proved to be a problem for testing as I had to decrease the number of boards that were run for each algorithm from 100 to 10 in order to get more timing results.

However, from the sizes that were run, the first version of the Stochastic Hill-Climbing with Random Restarts algorithm ran loads faster on average than the other local search algorithms. This makes sense as the algorithm is meant to restart with a new board in order to find a better solution, and with the stochastic 1 algorithm without the random restarts already being faster than the other algorithms (without random restarts), the restarts help the runtime and average iterations a lot.

The average conflicts for each run never went over 1, so the output never returned anything other than zero due to the nature of rounding.

Bonus: Implement a Genetic Algorithm for the n-Queens problem [+1 Point]

In [290]:

```
# Code and description go here
```