Solving the n-Queens Problem using Local Search

Instructions

Total Points: undergrad 10, graduate students 11

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

- your implementation (you can use libraries like math, numpy, scipy, but not libraries that implement inteligent agents or search algorithms),
- 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 $\mathbf{q} = \{q_1, q_2, \dots, q_n\}$, 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: $\mathrm{conflicts}(\mathbf{q})$
  >
  > 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 $\mathrm{conflicts}(\mathbf{q}^*) = 0$, however, the local improvement moves might end up in a local minimum.

Helper functions

In [2]:

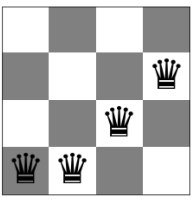
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors

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); // is 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 [3]:

```
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.
```



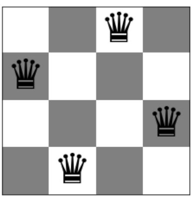
```
Queens (left to right) are at rows: [3 3 2 1]
Number of conflicts: 4
```

A board 4×4 with no conflicts:

In [4]:

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

```
Board with 0 conflicts.
```



Steepest-ascend Hill Climbing Search [3 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 [14]:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
import math
def random_board(n):
    #Creates a random board of size n x n
    return(np.random.randint(0,n, size = n))
```

In [15]:

```
def value(board):
    #Caclulate the number of conflicts
    board = np.array(board)
    n = len(board)
    val = 0
    for i in range(n):
        val += math.comb(np.sum(board == i), 2)
    # Check for each queen diagonally up and down
    for j in range(n):
        q_up = board[j]
        q_down = board[j]
        for jj in range(j+1, n):
            q_up -= 1
            q_down += 1
            if board[jj] == q_up: val += 1
            if board[jj] == q_down: val += 1
    return(val)
```

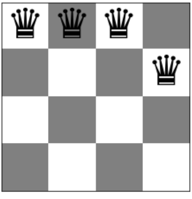
In [7]:

```
# Code and description go here
board = random_board(4)
def Steepest_Ascend_Hill_Cimbing(board, verbose = False):
    if verbose: show_board(board)
    n = len(board)
    best_val = value(board) # Current Conflict
    vals = np.full([n,n], -1, dtype = int)
    Number_of_steps = 0

    while True:

        for j in range(n):
            old_q = board[j]
            for i in range(n):
                board[j] = i
                vals[i,j] = value(board)
            board[j] = old_q
        new_min = np.min(vals)

        Number_of_steps = Number_of_steps +1

        #if verbose: print(vals)
        if verbose:
            print(f"Current Conflicts: {best_val} - New State Conflicts: {new_min}")
        if(best_val > new_min):
            w = np.where(vals == new_min)
            best = [a for a in zip(w[0], w[1])]
            best = best[np.random.randint(0, len(best))]
            board[best[1]] = best[0]
            best_val = new_min
            if verbose: show_board(board)
        else: return(board)
    print(Number_of_steps, "Number_of_steps")
```

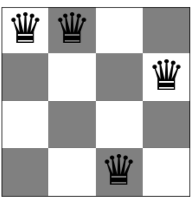
In [8]:

```
b = Steepest_Ascend_Hill_Cimbing(board, verbose = True)
show_board(b)
value(b)
```

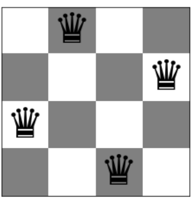
```
Board with 4 conflicts.
```



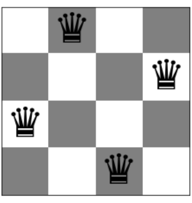
```
Current Conflicts: 4 - New State Conflicts: 1
Board with 1 conflicts.
```



```
Current Conflicts: 1 - New State Conflicts: 0
Board with 0 conflicts.
```



```
Current Conflicts: 0 - New State Conflicts: 0
Board with 0 conflicts.
```



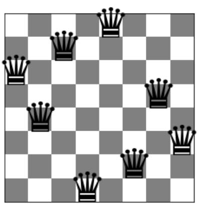
Out[8]:

```
0
```

In [9]:

```
def Steepest_Ascend_Hill_Cimbing_random_restarts(n, opt, restarts = 100, verbose = True):
    best_board = random_board(n)
    best_val = value(best_board)
    for i in range(restarts):
        board = random_board(n)
        board = opt(board, verbose = False)
        val = value(board)

        if val < best_val:
            best_board = board
            best_val = val
        if val == 0:
            if verbose: print (f"Needed restarts: {i+1}")
            return(board)
    if verbose: print(f"Could not find a solotion with {restarts} restarts.")
    return best_board
board = Steepest_Ascend_Hill_Cimbing_random_restarts(8, Steepest_Ascend_Hill_Cimbing)
show_board(board)
```

```
Needed restarts: 23
Board with 0 conflicts.
```



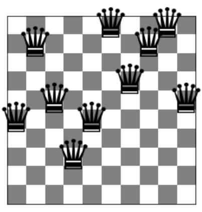
Stochastic Hill Climbing 1 [2 Point]

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).

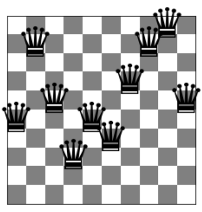
In [23]:

```
# Code and description go here
board = random_board(10)
def Stochastic_Hill_Climbing(board, verbose = False):
    #if verbose: show_board(board)
    n = len(board)
    best_val = value(board)
    vals = np.full([n,n], -1, dtype = int)

    while True:
    #Finding Uphill moves and choosing randomly from those moves
        for j in range(n):
            old_q = board[j]
            for i in range(n):
                board[j] = i
                vals[i,j] = value(board)
            board[j] = old_q

        Best_Uphills = []
        for i in range(n):
            for j in range(n):
                if vals[i][j] < best_val:
                    Best_Uphills.append(vals[i][j])
        if len(Best_Uphills) > 0:
            new_min = np.random.choice(Best_Uphills)

            #if verbose: print(vals)
            if verbose:
                print(f"Current Conflicts: {best_val} - New State Conflicts: {new_min}")
            if(best_val > new_min):
                w = np.where(vals == new_min)
                best = [a for a in zip(w[0], w[1])]
                best = best[np.random.randint(0, len(best))]
                board[best[1]] = best[0]
                best_val = new_min
                if verbose: show_board(board)
            else: return(board)
        elif Best_Uphills == []: break
    return(board)
b = Stochastic_Hill_Climbing(board, verbose = True)
show_board(b)
value(b)
```

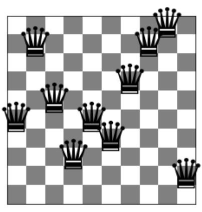
```
Current Conflicts: 10 - New State Conflicts: 8
Board with 8 conflicts.
```



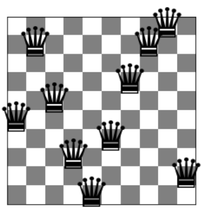
```
Current Conflicts: 8 - New State Conflicts: 6
Board with 6 conflicts.
```



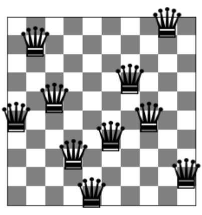
```
Current Conflicts: 6 - New State Conflicts: 5
Board with 5 conflicts.
```



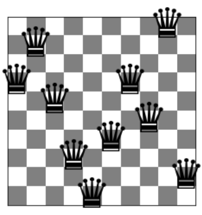
```
Current Conflicts: 5 - New State Conflicts: 3
Board with 3 conflicts.
```



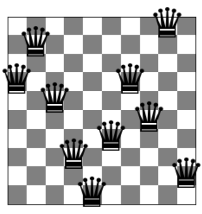
```
Current Conflicts: 3 - New State Conflicts: 2
Board with 2 conflicts.
```



```
Current Conflicts: 2 - New State Conflicts: 1
Board with 1 conflicts.
```



```
Board with 1 conflicts.
```



```
Out[23]: 1
```

Stochastic Hill Climbing 2 [2 Point]

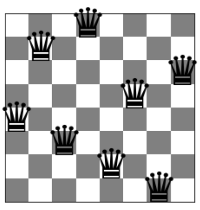
A popular version of stochastic hill climbing generates only a single random local neighbor at a time and accept it if it has a better objective function value than the current state. This is very efficient if each state has many possible successor states. This method is called "First-choice hill climbing" 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 [11]:
# Code and description go here
def Stochastic_Hill_Cimbing_random_restarts(n, opt, restarts = 100, verbose =True):
    best_board = random_board(n)
    best_val = value(best_board)
    for i in range(restarts):
        board = random_board(n)
        board = opt(board, verbose = False)
        val = value(board)

        if val < best_val:
            best_board = board
            best_val = val
        if val == 0:
            if verbose: print (f"Needed restarts: {i+1}")
            return(board)
    if verbose: print(f"Could not find a solotion with {restarts} restarts.")
    return best_board
board = Stochastic_Hill_Cimbing_random_restarts(8, Stochastic_Hill_Climbing)
show_board(board)
```

```
Needed restarts: 6
Board with 0 conflicts.
```

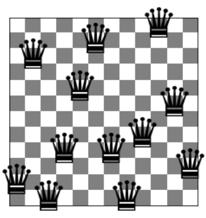


Hill Climbing Search with Random Restarts [1 Point]

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. Note that restart just means to run the algoithm several times with a new random initialization.

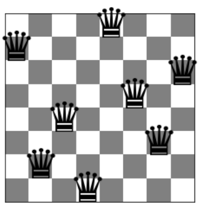
```
In [31]:
board = random_board(12)
def First_Choice_Hill_Climbing(board, verbose = False):
    #if verbose: show_board(board)
    n = len(board)
    best_val = value(board) # Current Conflict
    vals = np.full([n,n], -1, dtype = int)
    Number_of_steps = 0
    Not_Improve_steps = 0
    Limit = np.ceil(100*n)
    while Number_of_steps < 20000:


        for j in range(n):
            old_q = board[j]
            for i in range(n):
                board[j] = i
                vals[i,j] = value(board)
            board[j] = old_q
        random_number_one = np.random.randint(0,n)
        random_number_two = np.random.randint(0,n)
        new_min = vals[random_number_one][random_number_two]

        #if verbose: print(vals)
        if verbose: print(f"Current Conflicts: {best_val} - New State Conflicts: □→ {new_min}")
        Number_of_steps = Number_of_steps + 1
        print("Number of steps:", Number_of_steps, "\n" )
        if(best_val > new_min):
            Not_Improve_steps = 0
            w = np.where(vals == new_min)
            best = [a for a in zip(w[0], w[1])]
            best = best[np.random.randint(0, len(best))]
            board[best[1]] = best[0]
            best_val = new_min
            if verbose: show_board(board)
            if (best_val) == 0:
                break
        else:
            Not_Improve_steps = Not_Improve_steps + 1
            if Not_Improve_steps > Limit:
                break
        print("Nubmer of steps without improvment in a row:",␣ □→Not_Improve_steps, "\n")
        print(Number_of_steps, "Number_of_steps")
    return(board)
b = First_Choice_Hill_Climbing(board, verbose = True)
show_board(b)
```

```
Board with 3 conflicts.
```



```
In [32]:
def First_Choice_Hill_Climbing_random_restarts(n, opt, restarts = 100, verbose = True):
    best_board = random_board(n)
    best_val = value(best_board)
    for i in range(restarts):
        board = random_board(n)
        board = opt(board, verbose = False)
        val = value(board)

        if val < best_val:
            best_board = board
            best_val = val
        if val == 0:
            if verbose: print (f"Needed restarts: {i+1}")
            return(board)
    if verbose: print(f"Could not find a solotion with {restarts} restarts.")
    return best_board
board = First_Choice_Hill_Climbing_random_restarts(8, First_Choice_Hill_Climbing)
show_board(board)
```

```
Needed restarts: 4
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 [8]:
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: 4.4710612297058105 milliseconds
```

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

```
In [9]:
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: 4.978507979831193 milliseconds
```

Steepest-Ascend Hill Climbing

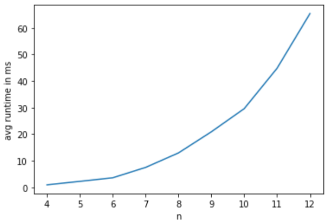
In [27]:

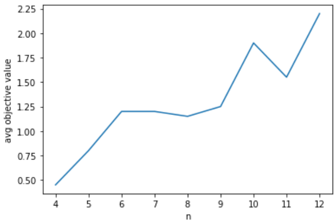
```
import time
N = 20
ns = [4, 5, 6, 7, 8, 9, 10, 11, 12]
mean_val = np.empty(len(ns), dtype = float)
mean_t = np.empty(len(ns), dtype = float)
for i,n in enumerate(ns):
    val = np.empty(N, dtype = int)
    t = np.empty(N, dtype = float)
    for j in range(N):
        t0 = time.time()
        board = random_board(n)
        sol = Steepest_Ascend_Hill_Cimbing(board)
        t1 = time.time()
        val[j] = value(sol)
        t[j] = t1 - t0
    mean_val[i] = np.average(val)
    mean_t[i] = np.average(t)
print(f"n = {n}: \tmean(time) = {round(mean_t[i]*1e3, 2)} ms\tmean(objective) = mean {mean_val[i]}" )
```

```
n = 12:         mean(time) = 65.36 ms   mean(objective) = mean 2.2
```

In [28]:

```
plt.plot(ns, mean_t*1e3)
plt.xlabel("n")
plt.ylabel("avg runtime in ms")
plt.show()
plt.plot(ns, mean_val)
plt.xlabel("n")
plt.ylabel("avg objective value")
plt.show()
```





Stochastic Hill Climbing

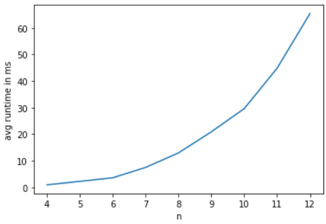
In [18]:

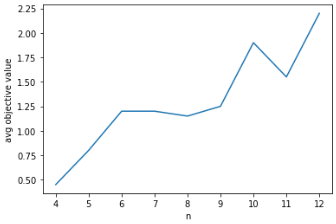
```
mean_val = np.empty(len(ns), dtype = float)
mean_t = np.empty(len(ns), dtype = float)
for i,n in enumerate(ns):
    val = np.empty(N, dtype = int)
    t = np.empty(N, dtype = float)
    for j in range(N):
        t0 = time.time()
        board = random_board(n)
        sol = First_Choice_Hill_Climbing(board)
        t1 = time.time()
        val[j] = value(sol)
        t[j] = t1 - t0
    mean_val[i] = np.average(val)
    mean_t[i] = np.average(t)
    print(f"n = {n}: \tmean(time) = {round(mean_t[i]*1e3, 2)} ms \tmean(objective) = mean {mean_val[i]}" )
```

```
n = 4:  mean(time) = 105.1 ms   mean(objective) = mean 1.05
n = 5:  mean(time) = 124.13 ms  mean(objective) = mean 0.45
n = 6:  mean(time) = 624.49 ms  mean(objective) = mean 0.95
n = 7:  mean(time) = 1359.49 ms         mean(objective) = mean 1.5
n = 8:  mean(time) = 2525.28 ms         mean(objective) = mean 1.3
n = 9:  mean(time) = 3395.98 ms         mean(objective) = mean 1.2
n = 10:         mean(time) = 7822.66 ms         mean(objective) = mean 1.45
n = 11:         mean(time) = 12678.52 ms        mean(objective) = mean 1.85
n = 12:         mean(time) = 16045.73 ms        mean(objective) = mean 1.25
```

In [29]:

```
plt.plot(ns, mean_t*1e3)
plt.xlabel("n")
plt.ylabel("avg runtime in ms")
plt.show()
plt.plot(ns, mean_val)
plt.xlabel("n")
plt.ylabel("avg objective value")
plt.show()
```





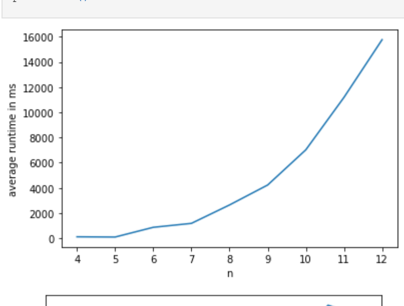
First Choice Hill Climbing

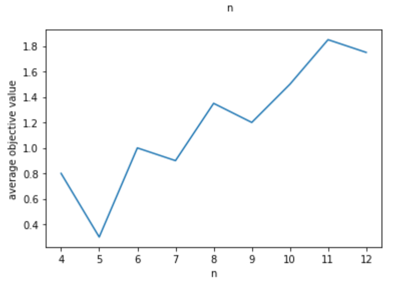
In [33]:

```
mean_val = np.empty(len(ns), dtype = float)
mean_t = np.empty(len(ns), dtype = float)
for i,n in enumerate(ns):
    val = np.empty(N, dtype = int)
    t = np.empty(N, dtype = float)
    for j in range(N):
        t0 = time.time()
        board = random_board(n)
        sol = First_Choice_Hill_Climbing(board)
        t1 = time.time()
        val[j] = value(sol)
        t[j] = t1 - t0
    mean_val[i] = np.average(val)
    mean_t[i] = np.average(t)
    print(f"n = {n}: \tmean(time) = {round(mean_t[i]*1e3, 2)} ms \tmean(objective) = mean {mean_val[i]}" )
```

```
n = 4:  mean(time) = 119.67 ms  mean(objective) = mean 0.8
n = 5:  mean(time) = 99.1 ms    mean(objective) = mean 0.3
n = 6:  mean(time) = 871.05 ms  mean(objective) = mean 1.0
n = 7:  mean(time) = 1184.02 ms        mean(objective) = mean 0.9
n = 8:  mean(time) = 2646.46 ms        mean(objective) = mean 1.35
n = 9:  mean(time) = 4230.2 ms  mean(objective) = mean 1.2
n = 10:         mean(time) = 7022.27 ms        mean(objective) = mean 1.5
n = 11:         mean(time) = 11183.79 ms       mean(objective) = mean 1.85
n = 12:         mean(time) = 15766.16 ms       mean(objective) = mean 1.75
```

In [34]:
```python
plt.plot(ns, mean_t*1e3)
plt.xlabel("n")
plt.ylabel("average runtime in ms")
plt.show()
plt.plot(ns, mean_val)
plt.xlabel("n")
plt.ylabel("average objective value")
plt.show()
```



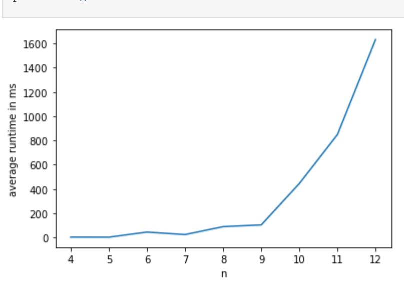


Steepest Ascend Hill Cimbing random restarts

In [36]:
```python
import time
N = 20
ns = [4, 5, 6, 7, 8, 9, 10, 11, 12]
RESTARTS = 100
mean_val = np.empty(len(ns), dtype = float)
mean_t = np.empty(len(ns), dtype = float)
for i,n in enumerate(ns):
    val = np.empty(N, dtype = int)
    t = np.empty(N, dtype = float)
    for j in range(N):
        t0 = time.time()
        sol = Steepest_Ascend_Hill_Cimbing_random_restarts(n, Steepest_Ascend_Hill_Cimbing, restarts = RESTARTS
        t1 = time.time()
        val[j] = value(sol)
        t[j] = t1 - t0
    mean_val[i] = np.average(val)
    mean_t[i] = np.average(t)
    print(f"n = {n}: \tmean(time) = {round(mean_t[i]*1e3, 2)} ms" )
```

```
n = 4:  mean(time) = 3.4 ms
n = 5:  mean(time) = 2.7 ms
n = 6:  mean(time) = 44.71 ms
n = 7:  mean(time) = 24.1 ms
n = 8:  mean(time) = 89.51 ms
n = 9:  mean(time) = 103.87 ms
n = 10:         mean(time) = 445.34 ms
n = 11:         mean(time) = 849.2 ms
n = 12:         mean(time) = 1632.69 ms
```

In [38]:
```python
plt.plot(ns, mean_t*1e3)
plt.xlabel("n")
plt.ylabel("average runtime in ms")
plt.show()
```

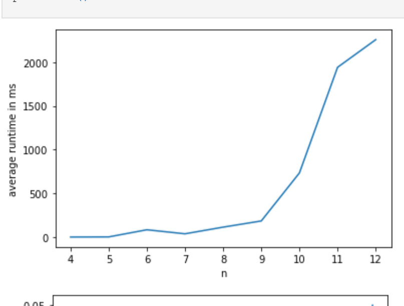


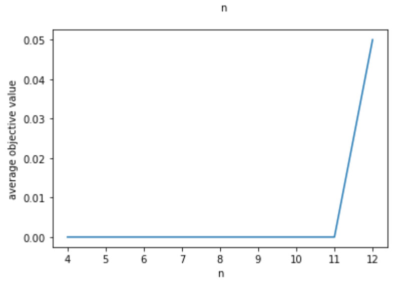
Stochastic Hill Cimbing random restarts

In [39]:
```python
RESTARTS = 100
mean_val = np.empty(len(ns), dtype = float)
mean_t = np.empty(len(ns), dtype = float)
for i,n in enumerate(ns):
    val = np.empty(N, dtype = int)
    t = np.empty(N, dtype = float)
    for j in range(N):
        t0 = time.time()
        sol = Stochastic_Hill_Cimbing_random_restarts(n,Stochastic_Hill_Climbing, restarts = RESTARTS, verbose
        t1 = time.time()
        val[j] = value(sol)
        t[j] = t1 - t0
    mean_val[i] = np.average(val)
    mean_t[i] = np.average(t)
    print(f"n = {n}: \tmean(time) = {round(mean_t[i]*1e3, 2)} ms\tmean(objective) = mean {mean_val[i]}" )
```

```
n = 4:  mean(time) = 3.41 ms    mean(objective) = mean 0.0
n = 5:  mean(time) = 4.99 ms    mean(objective) = mean 0.0
n = 6:  mean(time) = 86.41 ms   mean(objective) = mean 0.0
n = 7:  mean(time) = 40.38 ms   mean(objective) = mean 0.0
n = 8:  mean(time) = 116.39 ms  mean(objective) = mean 0.0
n = 9:  mean(time) = 186.95 ms  mean(objective) = mean 0.0
n = 10:         mean(time) = 734.88 ms  mean(objective) = mean 0.0
n = 11:         mean(time) = 1940.24 ms mean(objective) = mean 0.0
n = 12:         mean(time) = 2255.56 ms mean(objective) = mean 0.05
```

In [40]:
```python
plt.plot(ns, mean_t*1e3)
plt.xlabel("n")
plt.ylabel("average runtime in ms")
plt.show()
plt.plot(ns, mean_val)
plt.xlabel("n")
plt.ylabel("average objective value")
plt.show()
```



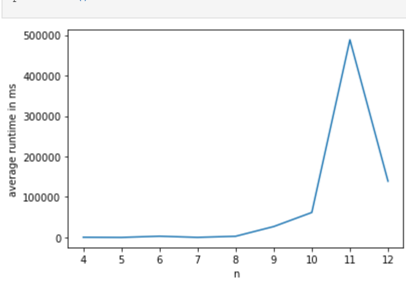


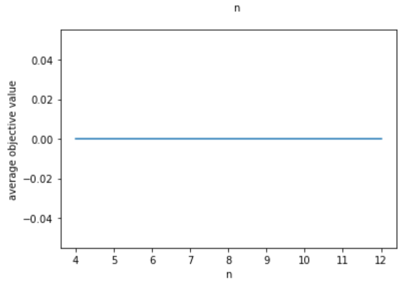
First Choice Hill Cimbing random restarts

In [50]:
```python
RESTARTS = 100
mean_val = np.empty(len(ns), dtype = float)
mean_t = np.empty(len(ns), dtype = float)
for i,n in enumerate(ns):
    val = np.empty(N, dtype = int)
    t = np.empty(N, dtype = float)
    for j in range(N):
        t0 = time.time()
        sol = First_Choice_Hill_Climbing_random_restarts(n, First_Choice_Hill_Climbing, restarts = RESTARTS, ve
        t1 = time.time()
        val[j] = value(sol)
        t[j] = t1 - t0
    mean_val[i] = np.average(val)
    mean_t[i] = np.average(t)
    print(f"n = {n}: \tmean(time) = {round(mean_t[i]*1e3, 2)} ms \tmean(objective) = mean {mean_val[i]}" )
```

```
n = 4:  mean(time) = 457.4 ms   mean(objective) = mean 0.0
n = 5:  mean(time) = 11.0 ms    mean(objective) = mean 0.0
n = 6:  mean(time) = 3010.5 ms  mean(objective) = mean 0.0
n = 7:  mean(time) = 127.0 ms   mean(objective) = mean 0.0
n = 8:  mean(time) = 2865.51 ms        mean(objective) = mean 0.0
n = 9:  mean(time) = 26883.74 ms       mean(objective) = mean 0.0
n = 10:         mean(time) = 61649.84 ms       mean(objective) = mean 0.0
n = 11:         mean(time) = 488505.04 ms      mean(objective) = mean 0.0
n = 12:         mean(time) = 138855.93 ms      mean(objective) = mean 0.0
```

In [51]:
```python
plt.plot(ns, mean_t*1e3)
plt.xlabel("n")
plt.ylabel("average runtime in ms")
plt.show()
plt.plot(ns, mean_val)
plt.xlabel("n")
plt.ylabel("average objective value")
plt.show()
```





Graduate student advanced task: Simulated Annealing [1 Point]

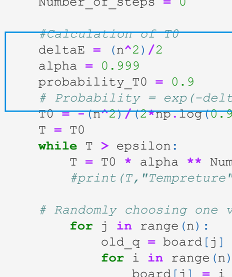
Undergraduate students: This is a bonus task you can attempt if you like [+1 Bonus point].

Simulated annealing is a form of stochastic hill climbing that avoids local optima by also allowing downhill moves with a probability proportional to a temperature. 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).

Implement simulated annealing for the n-Queens problem and compare its performance with the previous algorithms.

In [47]:
```python
[illegible]
show_board(b)
value(b)
```

Board with 0 conflicts.

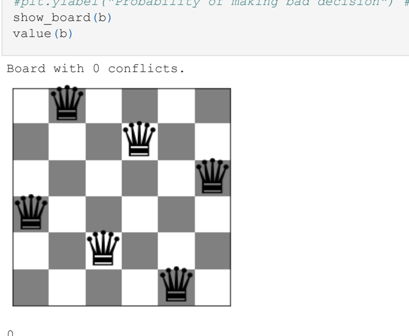


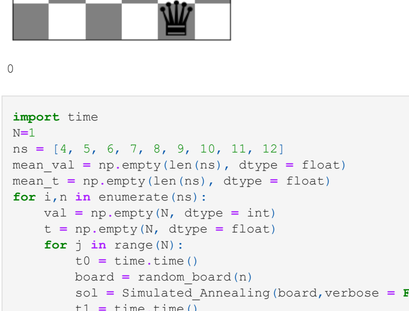
Out[47]: 0

In [48]:
```python
import time
N=1
ns = [4, 5, 6, 7, 8, 9, 10, 11, 12]
mean_val = np.empty(len(ns), dtype = float)
mean_t = np.empty(len(ns), dtype = float)
for i,n in enumerate(ns):
    val = np.empty(N, dtype = int)
    t = np.empty(N, dtype = float)
    for j in range(N):
        t0 = time.time()
        board = random_board(n)
        sol = Simulated_Annealing(board,verbose = False,T0 = None, alpha = 0.999, epsilon = 1e-1)
        t1 = time.time()
        val[j] = value(sol)
        t[j] = t1 - t0
    mean_val[i] = np.average(val)
    mean_t[i] = np.average(t)
    print(f"n = {n}: \tmean(time) = {round(mean_t[i]*1e3, 2)}  ms\tmean(objective) = mean {mean_val[i]}" )
```

```
n = 4:  mean(time) = 22.13  ms  mean(objective) = mean 0.0
n = 5:  mean(time) = 132.99  ms mean(objective) = mean 0.0
n = 6:  mean(time) = 2397.58  ms        mean(objective) = mean 0.0
n = 7:  mean(time) = 7865.43  ms        mean(objective) = mean 0.0
n = 8:  mean(time) = 21655.8  ms        mean(objective) = mean 0.0
n = 9:  mean(time) = 28827.22  ms       mean(objective) = mean 0.0
n = 10:         mean(time) = 47030.29  ms       mean(objective) = mean 0.0
n = 11:         mean(time) = 86571.27  ms       mean(objective) = mean 0.0
n = 12:         mean(time) = 96663.22  ms       mean(objective) = mean 0.0
```

In [49]:
```python
plt.plot(ns, mean_t*1e3)
plt.xlabel("n")
plt.ylabel("average runtime in ms")
plt.show()
plt.plot(ns, mean_val)
plt.xlabel("n")
plt.ylabel("average objective value")
plt.show()
```





More things to do

Implement a Genetic Algorithm for the n-Queens problem.

In [12]:
```python
# Code and description go here
```