1. The Big-Oh runtime for the brute force n-queens algorithm is O(n!)
2.
    a. The brute force algorithm takes 1.374 seconds when n=8
    b. The backtracking algorithm only takes .0003 seconds when n=8
    c. This makes the backtracking solution approximately 458 times faster for n=8

3. Backtracking is not always faster than brute force. When n is small, they take the same amount of time. However, as n grows the backtracking method is significantly faster. Here are the running times:

   Bruteforce (2)
   Running time: 0.0 ms

   Backtracking (2)
   Running time: 0.0 ms

   Bruteforce (3)
   Running time: 0.0 ms

   Backtracking (3)
   Running time: 0.0 ms

   Bruteforce (4)
   Running time: 0.0 ms

   Backtracking (4)
   Running time: 0.0 ms

   Bruteforce (5)
   Running time: 0.0 ms

   Backtracking (5)
   Running time: 0.0 ms

   Bruteforce (6)
   Running time: 13.999700546264648 ms

   Backtracking (6)
   Running time: 0.9868144989013672 ms

   Bruteforce (7)
   Running time: 39.99066352844238 ms

   Backtracking (7)
   Running time: 0.0 ms

   Bruteforce (8)

Running time: 1383.9564323425293 ms

Backtracking (8)
Running time: 2.997875213623047 ms

4. I will show that a square at position (i, j) is diagonal to a square at (x, y) if an only if $i + j == x + y$ or $i - j == x - y$ I will use the following definition of a diagonal square. Two squares (i, j) and (x, y) are diagonal if one of the following cases is true:

   $i - m = x$ and $j - m = y$

   $i - m = x$ and $j + m = y$

   $i + m = x$ and $j - m = y$

   $i + m = x$ and $j + m = y$

   I will show that each definition resolves to either $i + j == x + y$ or $i - j == x - y$
    a. For the first definition I will subtract the second formula from the first
        a. $(i - m = x) - (j - m = y)$
        b. $i - m - j + m = x - y$
        c. $i - j = x - y$
    b. For the second definition I will add both formulas together
        a. $(i - m = x) + (j + m = y)$
        b. $i - m + j + m = x + y$
        c. $i + j = x + y$
    c. For the third equation I will add both formulas together
        a. $(i + m = x) + (j - m = y)$
        b. $i + m + j - m = x + y$
        c. $i + j = x + y$
    d. For the fourth and final definition I will subtract the second formula from the first
        a. $(i + m = x) - (j + m = y)$
        b. $i + m - j - m = x - y$
        c. $i - j = x - y$

5. The Big-Oh run time for the fast_fib or iterative Fibonacci algorithm is O(n).
6.
    a. The fastest of the three Fibonacci algorithms implored is by far the matrix algorithm when referring to Big-Oh run times.
    b.
        i. Matrix Fibonacci n = 4 Running time: 0.0 ms
        ii. Fast Fibonacci n = 4 Running time: 0.0 ms
        iii. Recursive Fibonacci n = 4 Running time: 0.0 ms
        iv. Matrix Fibonacci n = 32 Running time: 0.0 ms
        v. Fast Fibonacci = 32 Running time: 0.0 ms
        vi. Recursive Fibonacci n = 32 Running time: 416.5472984313965 ms
        vii. Matrix Fibonacci n = 65536 Running time: 7.00068473815918 ms
        viii. Fast Fibonacci n = 65536 Running time: 51.44095420837402 ms

ix. Recursive Fibonacci n = 65536 Running time: N/A

c. All algorithms ran at the same speed when n = 4

d. At n = 32 the Matrix and Fast Fibonacci algorithms were the fastest

e. At n = 65536 the Matrix Fibonacci algorithm was the fastest

f. The fastest Big-Oh runtime was the Matrix Fibonacci algorithm with a runtime of O(log n). Big-Oh represents the worst scenario for each algorithm relative to the size of the input. The second fastest was the Fast Fibonacci algorithm with a runtime of O(n). The slowest algorithm was the Recursive Fibonacci algorithm with a runtime of O(2^n) which is on the slower side of Big-Oh runtimes.