Queens

Nicolas Dubreuil, Yann Koehnen

Summer Semester 2022

Supervising professors Gabor Wiese, Thierry Meyrath

Contents

1 Introduction **3**

2 N Queens Problem **4**
2.1 Eight Queens Problem 4
2.1.1 Backtracking method 4
2.1.2 Generating permutations 8
2.1.3 Python-code 9
2.1.4 Explanation of the code above 10
2.2 Number of Solutions 12
2.3 Explanation of the code above 14
2.4 Fundamental solutions 17

3 N+k Queens Problem **18**
3.1 Introduction 18
3.2 Python code 18
3.3 Explanation of the code 20

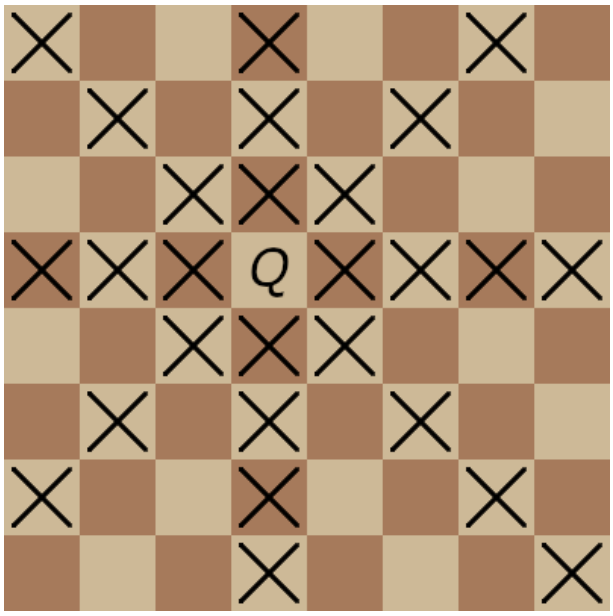
4 N Queens Problem on a Torus **24**
4.1 Python-code 28
4.2 Explanation of the code above 30
4.3 Number of Solutions on a Torus 31

5 Conclusion **32**

1 Introduction

There are a number of famous chess inspired mathematical puzzles. One of them is the queens puzzle, better known under the name eight queens puzzle. The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard in such a way that no two queens attack each other. A queen attacks all queens which share a row, column or diagonal with it. At the end of the introduction you will find a visualised attack pattern of a queen on an 8×8 chessboard.
We focus on specific variations of the queens problem, meaning we change the size and shape of the board. We also look at boards with holes. We will start by looking at the 8×8 case first to get familiar with the puzzle. Then we will introduce the N Queens Problem, where we try to place N queens on a $N \times N$ board. Furthermore, we will look at a special variation of the N queens problem, where we add k holes to the board and place $N + k$ queens. Each queen problem will be provided with a python-code and explanations on how we obtained various results. In the end, we will showcase the N queens problem on a Torus.

Attack pattern of a queen



Q: Queen
X: Field under attack

2 N Queens Problem

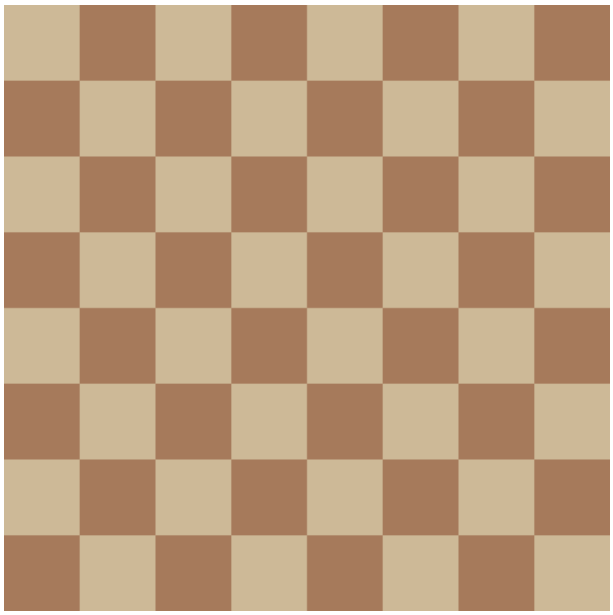
2.1 Eight Queens Problem

The eight queens problem is a simpler case of the more general N queens problem. For this puzzle, we look at an 8×8 board and we try to place eight queens without giving any queen an attack option.
There exist several techniques to solve this kind of problem. For example, one can simply try brute-forcing the problem, which will eventually lead to a solution. To be a bit more efficient, one can use the backtracking method. An even more efficient way to solve the problem is to create permutations. Using the backtracking and the permutation method will also make it easier to find more solutions.

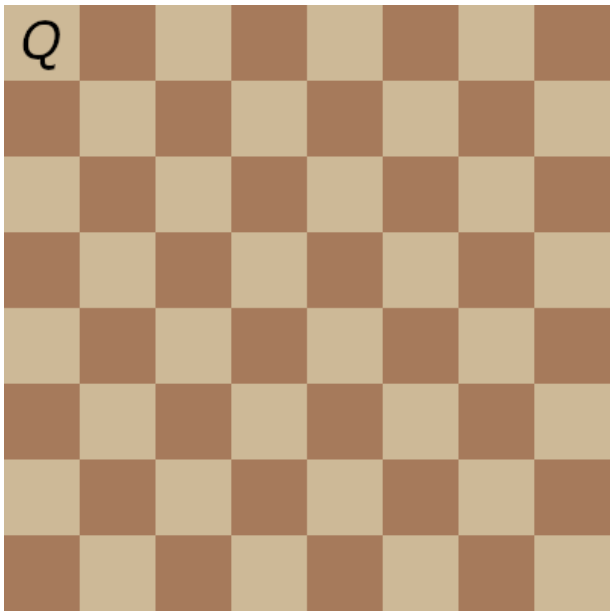
2.1.1 Backtracking method

The backtracking method considers searching every possible combination of queen placements to find a solution. The method works recursively, meaning it places one queen at a time and removes those that don't lead to a solution, while respecting the given constraints. With constraints, we mean that the 8×8 board must contain 8 queens, while no queen attacks another, to form a solution.
Illustrated example:
We imagine the board to be a coordinate system, where the origin is at the top left. We place the queens by assigning coordinates, meaning we use to numbers (x, y) to obtain their location on the board. The following seems counter intuitive, but x is the row-value, which indicates how far down the queen is from the origin, and y is the column-value, which indicates how far right the queen is from the origin. The origin has a coordinate of $(0, 0)$. For simplicity, we only use positive integers.

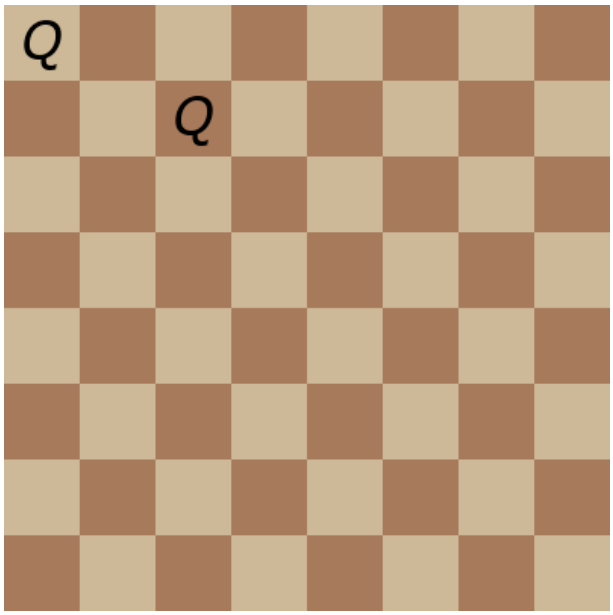
First we take an empty 8×8 board.



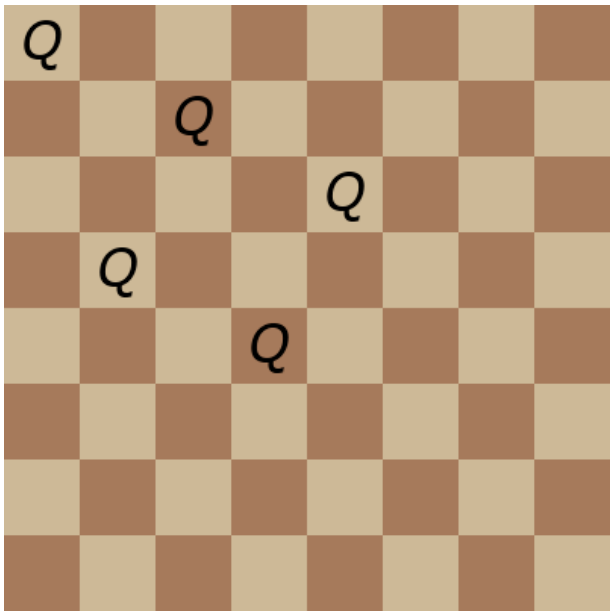
We place the first queen, represented as the symbol Q, at the coordinate $(0, 0)$.



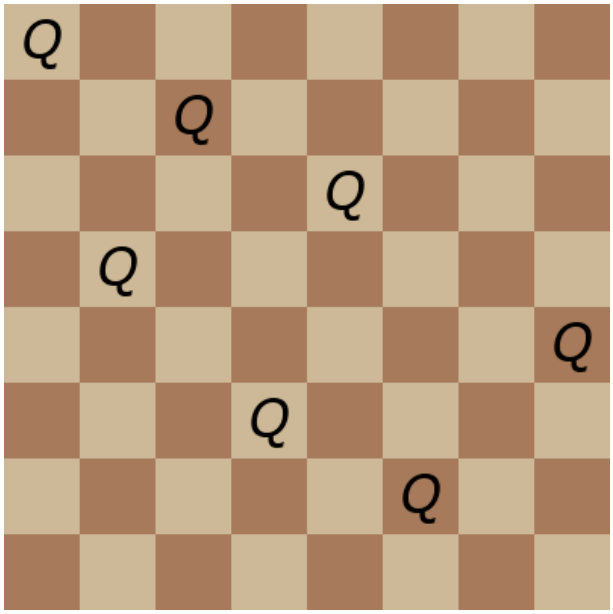
Now, we move on to the next row, since clearly the second queen cannot be placed in the same row as the first one. The next possible position for a queen will therefore be $(1, 2)$.



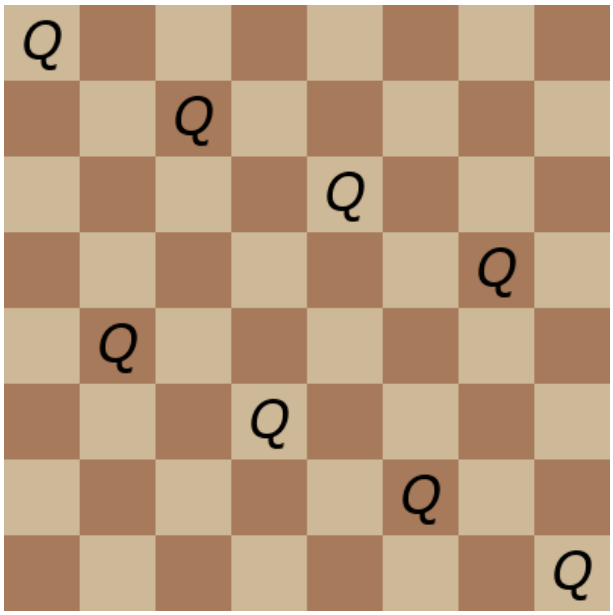
We repeat this process until we cannot place anymore queens.



Notice, it is impossible to place a queen in the 5^{th} row. Therefore we replace the last queen to the new position $(4, 7)$ and we continue to place queens at the first possible position.

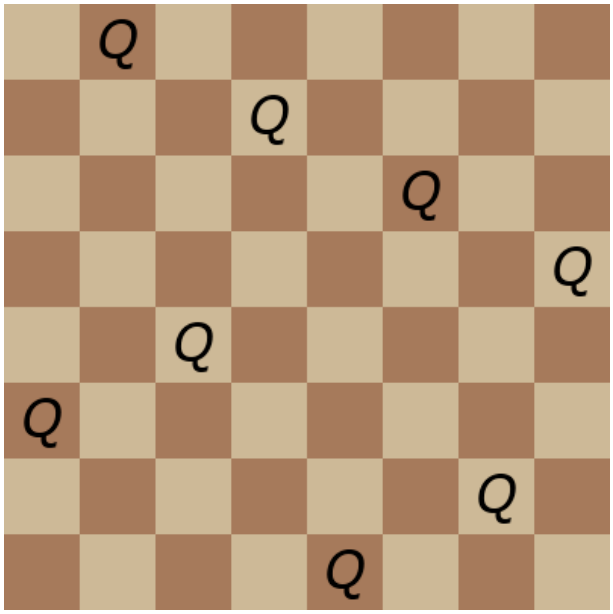


Again, it is impossible to place another queen. We backtrack once more. However, this time we need to go back to the placement of the 4^{th} queen and change its position to $(3, 6)$. We repeat the process and we finally obtain a solution.



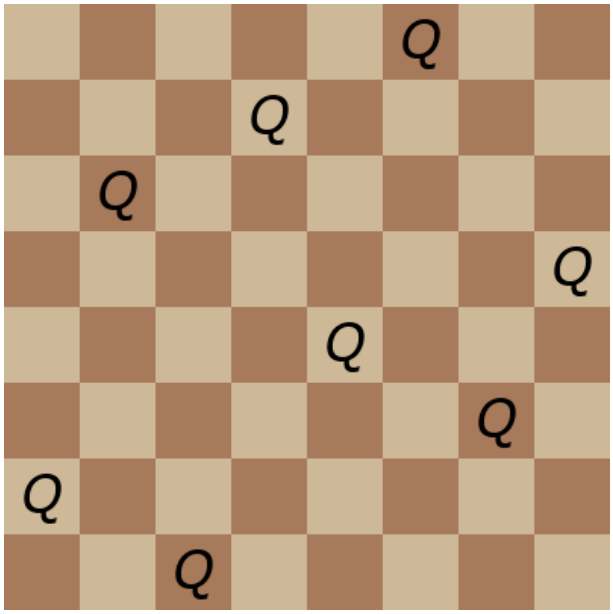
To obtain other solutions, one simply needs to check every possible option using the backtracking method. For example, changing the starting position gives another solution.

First Solution with starting Queen at $(0, 1)$



2.1.2 Generating permutations

We introduce a method, we call the permutation method, which takes the list $[0, 1, 2, 3, 4, 5, 6, 7]$ of 8 values and gives us every permutation of that list. Then we check for every permutation, if all 8 queens can be placed correctly. In other words, let's say we have the permutated list $[5, 3, 1, 7, 4, 6, 0, 2]$. Then the first queen would be placed at $(0, 5)$, the second at $(1, 3)$ and so on ... up to the last queen at $(7, 2)$. In this case the permutated list is a solution, since neither queen attack each other.



2.1.3 Python-code

Using the permutation method we wrote a Python program, which permits us to find all solutions for the N queens puzzle. The program is based on the permutation method, we explained in the previous section. As input, we take N, which gives us our $N \times N$ board size and the number of Queens, which will be placed on the board. The output will consist of the number of total solutions for specific N and it will provide us with a simple illustration of all the distinct solutions.
Program with simple explanations in form of comments [See [1]]:

```
from itertools import permutations
#board size and number of Queens
N = 8
#list [0, 1, 2, ..., N-1]
l = list(range(N))
#list of all the permutations of l
perms = permutations(l)
#list of N lists containing N zeros
board = [[0]*N for n in range(N)]
#setboard() will print an NxN board
def setboard():
    for r in range(N):
        print(board[r])
#placequeen(x, y) checks if a queen can be placed at (x, y).
#Thus no queen is allowed to be placed on the attack field of another
#or on the same field as another
#returns True if that is the case, False otherwise
def placequeen(x, y):
    if board[y][x] == 0:
        for m in range(N):
            board[y][m] = '_'
            board[m][x] = '_'
            board[y][x] = 'Q'
            #Attack pattern: diagonalBottomRight
            if y+m <= N-1 and x+m <= N-1:
                board[y+m][x+m] = '_'
            #Attack pattern: diagonalTopRight
            if y-m >= 0 and x+m <= N-1:
                board[y-m][x+m] = '_'
            #Attack pattern: diagonalBottomLeft
            if y+m <= N-1 and x-m >= 0:
                board[y+m][x-m] = '_'
            #Attack pattern: diagonalTopLeft
            if y-m >= 0 and x-m >= 0:
                board[y-m][x-m] = '_'
        return True
    else:
        return False
#counting the solutions
count = 0
#We use the permutation method
for p in perms:
    if all(placequeen(p[i],i) == True for i in range(N)):
        count += 1
        print('solution',count,':')
        setboard()
        print(' ')
    board = [[0] * N for n in range(N)]
```

2.1.4 Explanation of the code above

As already mentioned, the code takes N as the input. By changing the value of N, one changes the board size and the number of placed queens.
In the code, we use the inbuilt *permutations*() function, which provides us with all the permutations of a list. In our case, we call *perms* the list of all the permutations of the list called l. The list l is simply a list containing the integers from 0 to $N-1$. We define a list with $N \times N$ zeros and we call it *board*.
The function *setboard*() uses the list *board* and the value of N to print us an illustration that looks like a board.

Example of setboard() with $N = 8$

```
setboard()
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
```

Next we define the *placequeen*(x, y) function.

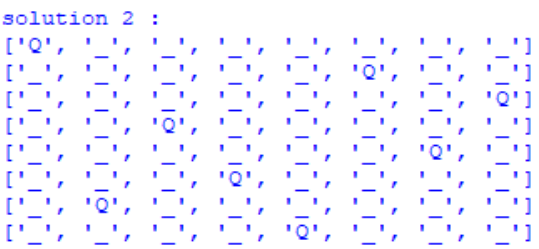
placequeen(x, y) function:
Input: x, y and the global variables N and *board*
Output: $True/False$
Description: The *placequeen*(x, y) function checks if a queen can be placed at (x, y) and returns $True/False$ accordingly. In addition, working with the variables N and *board*, we determine if a queen will be placed. A queen can only be placed, if the position in the board at which we call the function is equal to 0. If the function indeed decides that a queen will be placed, we place the symbol Q at the position. Then the function will also change every field to an underline, which would be under attack by the placed queen.

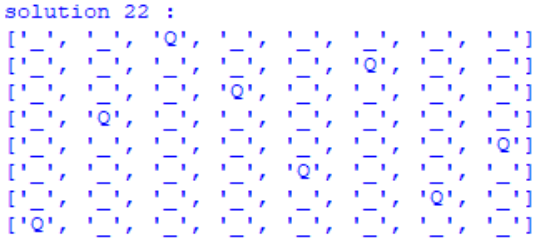
Example of placequeen()

```
placequeen(1, 0)
True
setboard()
['_', 'Q', '_', '_', '_', '_', '_', '_']
['_', '_', '_', 0, 0, 0, 0, 0]
[0, '_', 0, '_', 0, 0, 0, 0]
[0, '_', 0, 0, '_', 0, 0, 0]
[0, '_', 0, 0, 0, '_', 0, 0]
[0, '_', 0, 0, 0, 0, '_', 0]
[0, '_', 0, 0, 0, 0, 0, '_']
[0, '_', 0, 0, 0, 0, 0, 0]
placequeen(7,6)
False
```

These two functions will be used in the main part of the program, which is the permutation method.
main part:
Input: N
Description: We define $count = 0$, which will count how many solutions we will have for N. The main part checks for every single permutation p in the list $perms$, if the $placequeen$ function returns $True$ at $(p[i], i)$ for all values i between 0 and $N - 1$. In case that is true for a permutation a solution was found. Thus, we add 1 to $count$ and call $setboard()$ to print the solution, afterwards we reset the list $board$ such that we can repeat the process until we have gone through all the permutations of the list l.

Output examples





2.2 Number of Solutions

Using the permutation code, we found out that there are exactly 92 solutions for the eight queens problem. The table [4, Sequence A000170 and Sequence A002562] below confirms our claim and also shows the number of solutions for other N up to 27.

Number of Queens N	Total Solutions	Fundamental Solutions
1	1	1
2	0	0
3	0	0
4	2	1
5	10	2
6	4	1
7	40	6
8	92	12
9	352	46
10	724	92
11	2680	341
12	14200	1787
13	73712	9233
14	365596	45752
15	2279184	285053
16	14772512	1846955
17	95815104	11977939
18	666090624	83263591
19	4968057848	621012754
20	39029188884	4878666808
21	314666222712	39333324973
22	2691008701644	336376244042
23	24233937684440	3029242658210
24	227514171973736	28439272956934
25	2207893435808352	275986683743434
26	22317699616364044	278971246651089
27	234907967154122528	29363495934315694

The permutation code is not the fastest, when it comes to finding the number of solutions for higher values of N. This is the case, since the placequeen() function needs to run N times for every single permutation, even if for example the 3^{th} value of the permutation gives us a False statement. Since the all() function will only run if the placequeen() function checked all N values. Therefore, we introduce a new code, which we found online [See [7, Sample program]]. The following code is much faster in finding all the solutions for higher N. In addition this code is able to give us the fundamental solutions, which will be explained in detail in the next section.

Program [See [7, Sample program]] with simple explanations in form of comments:

```
N = 8
# Number of Queens on NxN chessboard
fsol = []
# List of all fundamental solutions
def RefH(board):
    return [[board[(N-1)-i][j] for j in range(N)] for i in range(N)]
# function used to reflect board along horizontal line
def Rot90(board):
    return [[board[(N-1)-j][i] for j in range(N)] for i in range(N)]
# function used to rotate board to 90 degrees
def Rot180(board):
    return Rot90(Rot90(board))

def Rot270(board):
    return Rot180(Rot90(board))

def RefHRot90(board):
    return RefH(Rot90(board))

def RefHRot180(board):
    return RefH(Rot180(board))

def RefHRot270(board):
    return RefH(Rot270(board))
# the five functions above use only the two first functions
board = [[0]*N for n in range(N)]
# define NxN board with "0" in all entries
def setboard():
    for r in range(N):
        print(board[r])
# function used to print all rows of board
def queens(n, i, a, b, c):
    if i < n:
        for j in range(n):
            if j not in a and i + j not in b and i - j not in c:
                yield from queens(n, i + 1, a + [j], b + [i + j], c + [i - j])
    else:
        yield a
```

```
# generator function used to find a solution
# i is the row
# a is used to check the column j
# b checks the / diagonal
# c checks the \ diagonal
count = 0
# define variable to count number of solutions
for solution in queens(N,0,[],[],[]):
# for-loop with every different solution of function queens
    for i in range(N):
        board[i][solution[i]] = 1
# "solution" is a list with elements denoting positions of queens
# we change those positions in board to "1"
    sym = (board,Rot90(board),Rot180(board),Rot270(board),
           RefH(board),RefHRot90(board),RefHRot180(board),RefHRot270(board))
# create tuple "sym" where we have all possible rotations/reflexions
# of current board
    new = True
    for k in range(len(sym)):
        if (sym[k] in fsol):
            new = False
# for every element of "sym", if any already in "fsol"
# we don't have a new solution
    if new:
# if solution is new, we add it to list of solutions "fsol"
        fsol.extend([board])
        count += 1
        print('solution',count,':')
        setboard()
        print(' ')
# we print the result
    board = [[0] * N for n in range(N)]
# reset board for next solution

# gives us the total number of solutions
def TotalNumberOfSolutions():
    totalcount = 0
    for solution in queens(N, 0, [], [], []):
        totalcount += 1
    print('Total number of solutions:', totalcount)
```

2.3 Explanation of the code above

This code works again for any N we give as input. This N is the number of queens we want to place on a $N \times N$ chess board. We define an empty list called $fsol$ where we will store all the fundamental solutions later on.
Next we define the functions $RefH()$ and $Rot90()$.

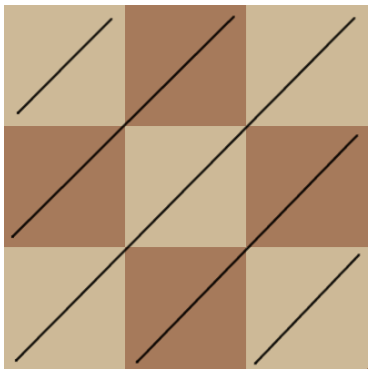
$RefH()$:
Input: *board*
Output: returns a reflection of the *board* along the horizontal line
Description: returns a list where the new rows of the reflected board are lists themselves and elements of the returned list. We start at the last row and we get the elements of that row as a list, which will be the first element of the returned list. We repeat until the first row. In other words, we inverse the order of the rows.

$Rot90()$:
Input: *board*
Output: returns a 90 degrees rotation of the *board*
Description: returns a list of the rows obtained after a 90 degrees clockwise rotation. The

first new row will be the first old column but in inverse order. The same goes for the second, third,..., until the last.

Note that these two functions are enough to represent all the rotations and reflections in a square board. In fact we define the following new functions: $Rot180()$, $Rot270()$, $RefHRot90()$, $RefHRot180()$ and $RefHRot270()$. These functions do exactly as their name suggests, for instance $RefHRot90()$ returns the board after first rotating it 90 degrees to the right, and then reflecting it. Hence, we can call upon the first two functions we defined to obtain such a result. The rest of the functions work in a similar manner.
The board we used in the functions above has to be defined first of course. The *board* is a list with N lists where each list contains N elements equal to 0. The smaller lists inside the *board* list represent the rows of the board. Furthermore we define the function $setboard()$, which prints every row of the board in a way that the output resembles a real chessboard.
Now we define the most important function, the $queens()$ function. In fact, in python it is known as a generator function. Instead of returning, we yield. Put simply, contrary to the return statement, the yield statement does not stop the function execution entirely, but saves the state of the function and returns the yielded value to the caller.

$queens()$:
Input: n, i, a, b, c
Explanation of the input: The parameter n is simply the number of queens N we want on our $N \times N$ board. The value i represent the current row of the board. The parameter a is a list that helps us to check the column j of our current row i. The parameters b and c are also lists that help us to check the current / and \ diagonals respectively.
Output: yield a
Description: We first check if our current row i is smaller than n ($n-1$ is the last row). If that is the case, for every j with j going from 0 to $n-1$, we check if j is not in a and $i+j$ is not in b and $i-j$ is not in c. The list a starts out empty and we only add the value of j to it if it wasn't there before (and if the b and c conditions are also met), hence j not being in a means that the column j is free. Next we look at the list b, which also starts out empty. The sum of the coordinates of a point on a board is the same for every point on a single / diagonal. In fact, the sums can go from 0 to $(n-1)+(n-1) = 2n-2$. To understand, consider a 3×3 board. The points at $(0,0)$ will have value 0, the points at $(0,1),(1,0)$ will have value 1, the points at $(0,2),(1,1),(2,0)$ will have value 2, the points at $(1,2),(2,1)$ will have value 3 and the points at $(2,2)$ will have value 4. This is the case for any N.



Hence saying that $i+j$ is not in b means that the / diagonal is clear. It is similar for c. The difference of the coordinates $i-j$ of a point on a board is the same for every point on a single \ diagonal. Indeed, the differences go from 0 to $-(n-1)$. The idea is the same as for b, hence to say that $i-j$ is not in c means that the \ diagonal is clear. Now if these three conditions are met, we yield from queens$(n, i+1, a+[j], b+[i+j], c+[i-j])$, where we went to the next row and values that were not in a, b, c are added to their respective lists. This process is repeated until $i < n$ is not true anymore. This means that all the columns of the board have one single queen in them. We yield the list a, because the list gives us exactly in which columns the queens are placed for each row (the first element in a is the column in row 0, ..., the last element in a is the column in row $n-1$).

Next we define the variable count to be zero as we use it to count the number of solution.

main part:
Input: N and the variables i, a, b, c are assigned to $0, [], [], []$ respectively.
Description: We create a for-loop. We consider every solution in $queens(N, 0, [], [], [])$. This will give us all the different lists a that were yielded from the function. So for every such solution, we replace by 1 the value in every position of the board where a queen should be placed. After that, we create a tuple we call sym where we include all eight variants a board can have (the board itself, the board after a $90, 180$ and 270 degrees rotation and the reflection of each of the four). Let us now define a variable called new which is set to $True$. Next for every element in the tuple sym, if any of them is in the list $fsol$, then that means that there is already a version of the board that can be obtained by applying rotation and reflection operations on our current board. In that case we set the variable new to be $False$. The only way for the variable new to stay $True$, is that none of the variants of the current board in sym already are in the list $fsol$. Then in the case that new is indeed $True$, we add the current board to the list of fundamental solutions. Then add value 1 to the variable $count$, and print the current board with the function $setboard()$. After the if-statement, it is important to reset the board in order for the next solution to be processed correctly.

This part of the code only gives us the fundamental solutions. The next function will provide us with all solutions. We define the $TotalNumberOfSolutions()$ function.

$TotalNumberOfSolutions()$:
Input: N
Description: Gives us the total number of solutions for N by simply adding 1 to the beforehand defined parameter $totalcount$ for every solution in queens$(N, 0, [], [], [])$ and then printing $totalcount$ after the for-loop ended.

2.4 Fundamental solutions

As one might have already noticed, we have already given the precise number of the fundamental solutions for N up to 27 in the table of the section 3.1. But what exactly are fundamental solutions? Let's look at the case where $N = 8$. Then we have 92 distinct solutions. Now if we take the symmetry operations of rotation and reflection of the board of the solutions into account, then if we count these solutions as one, we will only have 12 solutions left, which will be called fundamental solutions. Most of the time a fundamental solution has eight forms. First of all we have its original form, which we can subsequently rotate three times by 90 degrees. Meaning, we rotate the original form by 90, 18 and 270 degrees, so we currently know about the existence of four forms. Secondly, we can now take the reflection of each of these four forms, to obtain eight forms in total.
Output of the new code:

```
solution 1 :
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]

solution 2 :
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]

solution 3 :
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 1, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0]

solution 4 :
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]

solution 5 :
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]

solution 6 :
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]

solution 7 :
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 1, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0]

solution 8 :
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]

solution 9 :
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]

solution 10 :
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]

solution 11 :
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 1, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]

solution 12 :
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

Notice that solution 10 in the output has only four forms in total, since it is identical to its own 180 degrees rotation. The four forms consist of its original form and its reflection and of the original form's 90 degrees rotation and its reflection. Therefore, the total number of distinct solutions for $N = 8$ is $11 \cdot 8 + 1 \cdot 4 = 92$.

3 N+k Queens Problem

3.1 Introduction

Let us now introduce the $N + k$ queens problem. We still work on a $N \times N$ chessboard, but we now wish to place $N + k$ queens on such a board without any attacking each other. However simply placing $N + k$ queens on a $N \times N$ board doesn't sound possible, which is why we also add k holes to the board. Note that a hole could also be considered as a pawn, as we define that a queen cannot attack over a hole. In other words, it would for instance be possible for two queens to be in the same row if they are separated by a hole. In short, the goal is to place $N + k$ queens and k holes such that no two queens can attack each other.

Let us state some important results.

Theorem 3.1 (N+1 Queens)**.** *[See [8, Theorem 1]] For $N \geq 6$, there is a solution to the $N + 1$ queens problem.*

Conjecture 3.2 (N+k Queens)**.** [See [8, Conjecture 2]] For any positive integer k and large enough N, there is a solution to the $N + k$ queens problem.

Remark 3.3. The Conjecture above could also be stated as a Theorem. In fact the program that we will present shortly shows that it is indeed true for smaller values of N, but we can only assume the same for larger values of N.

In fact, we should have the following number of total solutions [See [3]]

N	k = 1	k = 2	k = 3
6	16	0	0
7	20	4	0
8	128	44	8
9	396	280	44
10	2288	1304	528

and of fundamental solutions

N	k = 1	k = 2	k = 3
6	2	0	0
7	3	1	0
8	16	6	1
9	52	37	6
10	286	164	66

.

3.2 Python code

Now we introduce a python code [See [2]], where we place $N + k$ queens on a $N \times N$ board with k holes in it. The goal is to find the number of solutions for the $N + k$ problem where N and k is given as input. Note that the holes are not placed at random but in such a way to find all possible solutions.

```
#function checks direction (dx,dy) for hole or edge of board
#return True if we meet hole or edge, False if queen
def Direction(r,cq,dx,dy):
    #x goes in direction indicated by dx
    x = r + dx
    #y goes in direction indicated by dy
    y = cq + dy
    while (x>=0) and (y>=0) and (x<N) and (y<N):
        #2 indicates a hole
        if board[x][y]==2:
            return True
        #1 indicates a queen
        if board[x][y]==1:
            return False
        #we add direction to x and y
        x = x + dx
        y = y + dy
    return True
#function checks if queen is safe to place by looking at directions
#up, left, top left and top right
def isSafe(r,cq):
    return (Direction(r,cq,-1,0) and Direction(r,cq,0,-1) and Direction(r,cq,-1,-1)
            and Direction(r,cq,-1,1))
#function to check what the last piece in column is
#return 1 if a queen
#return 2 if a hole
#return 0 if nothing (empty column)
def LPC(ch):
    i = N-1
    #we check current column from down bottom to top
    while (i>=0):
        if board[i][ch]!=0:
            return board[i][ch]
        i = i-1
    return 0

#function to place queens and holes
def QAH(r,c,h):
    #if row r exceeds board(0 to N-1) and no holes h to place
    #call function that outputs board solution
    if (r==N) and (h==0):
        Solution(board)
    else:
    #we check for each column between c and N-1 included in current row
        for cq in range(c,N):
    #if position is not attacked from any direction we place queen
            if isSafe(r,cq):
                # 1 denotes queen
                board[r][cq]=1
                #if there are still holes to place, we check the columns
                #to the right of queen in current row
                if h > 0:
                    #hole must be between two queens
                    for ch in range(cq+1,N-1):
                #if last piece in column is queen, we put hole in current
                #position and look for all solutions with current hole
                        if LPC(ch)==1:
                            board[r][ch] = 2
                            QAH(r,ch+1,h-1)
                        #we remove hole and check rest of row as next
                        #for-loop iteration begins
                            board[r][ch] = 0
                #if no more holes to place or for-loop over
                #then check next row and find other solutions
                #with queen in position (r,cq)
                QAH(r+1,0,h)
                #all solutions with queen in position (r,cq) found
                #remove queen from position and check for other solutions
                board[r][cq] = 0
```

```
#function that that replaces 0,1 and 2 by *,Q and H
#prints the board for each solution
def Solution(board):
    global allsol
    #add solution to list of all solutions
    allsol.extend([board])
    print("Solution ",len(allsol),":")
    for i in range(N):
        for j in range(N):
            if board[i][j] == 1:
                print('Q', end=' ')
            if board[i][j] == 0:
                print('*', end=' ')
            if board[i][j] == 2:
                print('H', end=' ')
        print(' ', end='\n')
    print(' ', end='\n')

#define variables to be global
global board, allsol, N
N = 8
h = 3
#define the board with 0 at every point
board = [[0 for i in range(N)] for j in range(N)]
#define empty list of all solutions
allsol=[]
#Call function with desired input
QAH(0,0,h)
#print total number of solutions
print("Number of solutions:",len(allsol))
```

3.3 Explanation of the code

We first define the function $Direction()$. The function is used to determine what piece is placed in a certain given direction from a given position. The possibilities are a hole, a queen or nothing, meaning that we meet the board's edge.

$Direction()$:
Input: r, cq, dx, dy
Explanation of the input: The value r is to determine the current row of the board (note that the rows are denoted from 0 to $N-1$). The value cq denotes the current column where a queen could be placed. The values dx (rows) and dy (columns) represent the direction in which we look. They can take the values $-1, 0$ and 1. Since the board is expressed in form of a list with multiple lists inside that represent the rows, $(0, 0)$ is the top left position of the board (board[0][0]). Hence we can say that for dx, dy being 0, the position does not move. For $dy = 1$, the columns move to the right and for $dy = -1$ the columns move to the left. For $dx = 1$ the rows move down and for $dx = -1$, the rows move up.
Output: $True/False$
Description: When the function is called, we define two variables x, y to be equal to the row r and column cq with their respective direction added to it. Then we create a while loop that will be active as long as x and y are both between 0 and $N-1$. For every iteration of the loop, we check if the position we have moved to is occupied by either a queen (then we return $False$ because that would mean that it is not safe to place queen at position given at the beginning of the function) or a hole (then we return $True$ because a queen would be safe to place). However, if we find neither a hole nor a queen, we add the directions dx, dy to x, y again and start the next iteration. If even after every iteration of the loop nothing has been returned, we return $True$ because it means we met the edge of the board.

Next we define the function $isSafe()$.

Input: r, cq
Output: $True/False$
Description: Here we call upon the function $Direction()$ for the directions up ($dx = -1$ and $dy = 0$), to the left ($dx = 0$ and $dy = -1$), to the top left ($dx = -1$ and $dy = -1$) and to the top right ($dx = -1$ and $dy = 1$). If the function $Direction()$ returns $True$ for every of the four given directions, then the function $isSafe()$ returns $True$ and the queen is indeed safe to place in the current position.

As we will see later, when running the code we run through every element of a row before going to the next column. This means that we only need to check the positions behind the current position and not the ones that come after. Now we define the function $LPC()$ where we check what the last piece in the given column is.

$LPC()$:
Input: ch
Explanation of the input: ch indicates the column of a possible future hole, as we will see later.
Output: returns $1, 2$ or 0
Description: In the given column ch, we start in the last row (hence $i = N - 1$) and while that i is still greater or equal to 0 (meaning row is still part of the board), we check what the piece in that position is. We stop the loop as soon as we come upon a piece (hole or queen). That is if the current position is not empty (empty denoted by 0 in current position), then we return the value of the position. If the value is 1, we know it's a queen. If it is 2, we know it is a hole. However if for every position in the given column, the position is empty, then the function returns 0 because the entire column will be empty.

We now come to the most important function of this code. We define the function $QAH()$.
Input: r, c, h
Explanation of the input: The parameters r and c indicate the rows and columns and h denotes how many holes are left to place on the board.
Output: solutions
Description: As previously mentioned, when run, the code goes from one row to the next. Hence if the function were to be called with the row $r = N$ (note that rows go from 0 to $N - 1$) as parameter and $h = 0$ (which means that there are no holes left to place), then we have a complete board and will can upon the function $Solution()$ with the current board as its parameter to output the solution. However if that is not the case, then our board is not yet complete. At this point, we are still at the position indicated by r and c, but since we go through one row after another, we check the rest of the columns in our current row r. We create a for loop that considers all the columns cq between our starting column c and the last column $N - 1$ to place a queen. For such a column cq and row r, we call upon the function $isSafe()$. As explained before, if it returns $True$, it means that no position before our current position prevents a queen from being placed in the current position with row r and column cq. Hence we set that position in the board to be equal to 1 (which denotes a queen). Next we check if there are still holes to place i.e if $h > 0$. If so we have multiple things to do. First it is important to note that in rows with holes, the holes and queens alternate. This means that a hole is always between two queens.

Now we check every column ch (still in row r) after the column where we placed the last queen until the second last column (it cannot be the last column because then it would not be possible to have a queen to the right of the hole). For such a column, we call upon the function $LPC()$, which will return what piece is the last piece in that column. Note that the hole has not yet been placed, and since holes and queens alternate, the first piece above our current position would need to be a queen in order for a hole to be placed. If the function $LPC()$ indeed returns 1, then we can place the hole in the current position with row r and column ch (we place a 2 in that position of the board). However it is now important for a queen to be placed to the right of the hole (still in row r), hence we call upon the function $QAH()$ itself but with parameters $r, ch + 1, h - 1$. If a queen can indeed be placed to the right of the hole, then the code continues as intended and tries to place the next queens and holes. In fact we look for all the solutions with a hole at current position. But if there is no possible way to place a queen to the right of the hole, then the hole is not allowed to be there and hence we delete it by resetting its position to 0. The for loop continues and repeats this process. The next line of code is reached in two possible ways. Either the for loop is over or the condition $h > 0$ failed to be $True$. In any case we call again upon the function $QAH()$ itself and we go to the next row. To be precise, the function is called with the parameters $r + 1, 0$ and h. Now we look for all the other solution with a queen in the position row r and column cq. As soon as we have found all the solutions with a queen in that position, we reset the position (a 0 in the position of the board) and we look for solutions with this position being empty. That is, the first for loop continues. The function calls itself until every possible solution has been found.

It might be a bit hard to understand how multiple solutions can be printed when the function $QAH()$ is over once the first if statement is true. In short, when the function runs, at some point it will call upon itself, then during that process it might call upon itself again and so on. The important thing to remember is that when the first if statement of the function is true, then the execution of that function ends, but the function that called the function that just ended continues. This will repeat multiple times until every possible solution has been printed.

Finally we have the last function of the code. We define the function $Solution()$.
Input: $board$ and the global variable $allsol$
Explanation of the input: $allsol$ is the list where each board (solution) is added to
Output: Prints the solution board for us.
Description: For the solution given as a parameter, we print some characters according to the values of the board. As often mentioned, when a position is equal to 1, we print a "Q" denoting a queen. When the value is 2, we print "H" for a hole, and when it is 0, we print "*" denoting an empty space. We also print some phrases and spaces to make it more aesthetically pleasing.

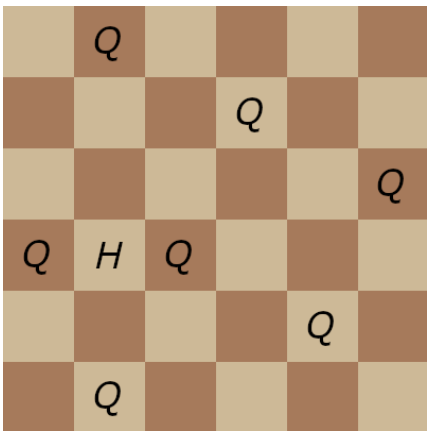
Now for the final details, we define the global variables $board, allsol$ and N. N denotes the value for our $N \times N$ board and the variable $board$ represents the board itself. We set the values for N and h, and we will have $N + h$ queens on the board in the end. We also set the initial value for $board$ which is a list where the elements are also lists representing the rows and $N - 1$ zero's in every such list (row). The variable $allsol$ is also set as an empty list. We now call the $QAH()$ function beginning at position $(0, 0)$ and h holes. We also print the number of total solutions.

Output examples: 8+3 and 7+2 Queens

```
Solution  3 :
* * * Q * * * *
* Q * * * * * *
* * * * Q * * *
* * Q H * * * Q
Q H * * * Q * *
* * * Q * * * *
* Q * * H * Q *
* * * * Q * * *
```

```
Solution  1 :
* * Q * * * * *
* * * * Q * * *
* Q H * * * Q *
* * * Q * * * *
Q * * * H Q * *
* * Q * * * * *
* * * * Q * * *
```

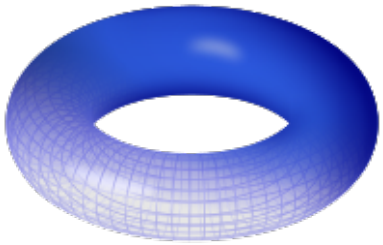
6+1 Queens



4 N Queens Problem on a Torus

In this section, we will analyse the N-Queens problem on a chessboard covering a torus. Here is a little reminder of what a torus looks like.

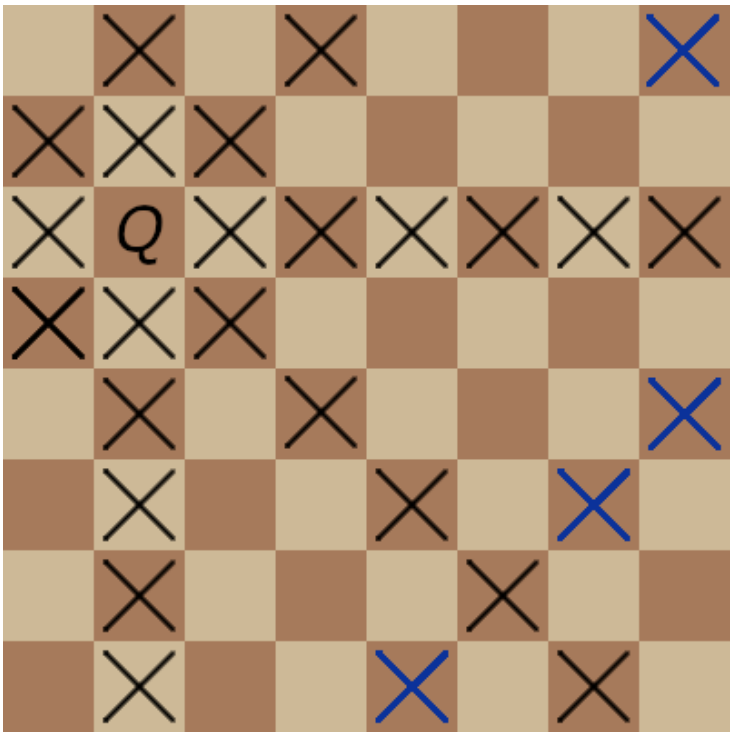
Figure 1: picture found at [6]



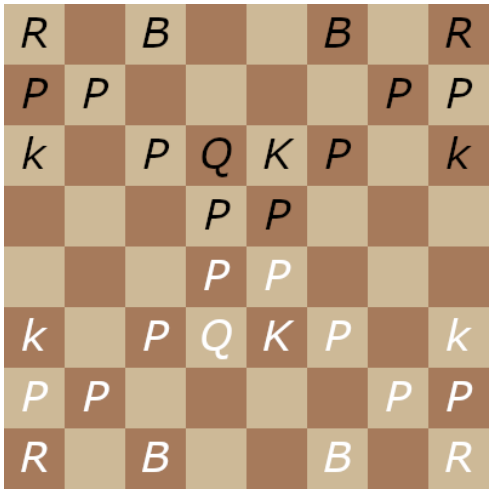
We now introduce toroidal chess. The rules of chess on a torus are different yet fairly similar to normal chess. First of all, how do we represent the board? It might be best to first imagine the concept of cylindrical chess. As the name suggests, it is played as if the board were a cylinder. This is how it looks.

Figure 2: picture found at [5]

However we can represent it on a normal $2D$ chessboard. The moves are the same as in the normal version, with the difference being that the left and right sides of the board aren't endpoints. Indeed one can move to the left and come out on the right side and vice-versa. Now that we know how chess on a cylinder works, we can take the next step to chess on a torus. We take the cylinder in figure 2 and imagine we connected the flat surfaces of the cylinder. The result would look like a doughnut.
In fact, this can also be represented in a $2D$ chessboard. In addition to the left and right sides being connected, moves from top side to bottom side and vice-versa are also possible. The possible moves of a queen chess piece are being made visually clear in the following figure.



Remark 4.1 (Fun Fact). Though not relevant to the topic at hand, this is the starting position of torus chess



Q: Queen
K: King
P: Pawn
k: Knight
R: Rook
B: Bishop

Definition 4.2. The N-Queens problem on a torus is the mod N-Queens problem.

The following example makes the definition a bit more clear.

Example 4.3. Consider a 5×5 board and imagine a queen were placed on the square $(3, 3)$ and it wanted to attack a queen on the square $(0, 1)$ (which is possible on a torus). The queen would simply move diagonally (direction bottom left) for 2 squares. In fact $(3 + k, 3 - k) \pmod 5$ for $k = 2$ is exactly the position $(0, 1)$.

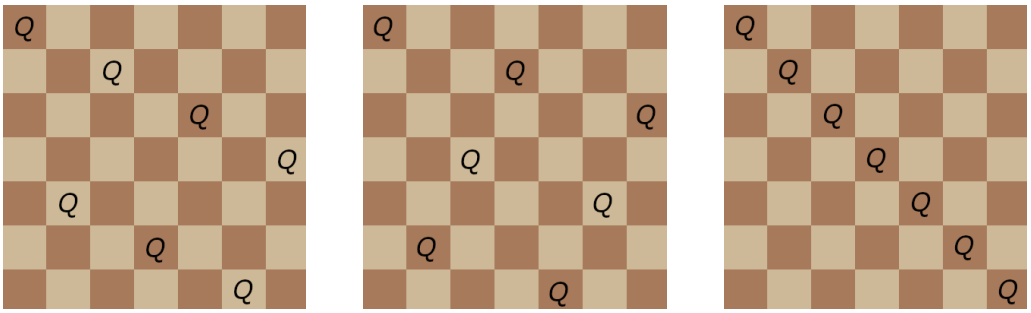
As we look at the mod N-Queens problem, with the squares of the board seen as coordinates ($(0, 0)$ top left and $(N - 1, N - 1)$ bottom right), for a solution we need a permutation σ of $(0, \ldots, N - 1)$ such that the queens in positions $(i, \sigma(i))$ do not attack each other for $i = 0, \ldots, N-1$. In fact, if a queen were to be placed on square $(i, \sigma(i))$, then that queen attack every square in its row and column, but also the squares $(i+k, \sigma(i)+k) \pmod N$ and $(i + k, \sigma(i) - k) \pmod N$ for any k. Hence we get that $\sigma + x$ and $\sigma - x$ are also permutations if σ itself is a solution. We define x to be the identity permutation (sends i to i).

Before we show how to compute how to find solutions for the N-Queens problem on a torus, let us state an important theorem.

Theorem 4.4 (See [9, Theorem 1])**.** *There exists a solution for the mod N-Queens problem if and only if $gcd(N, 6) = 1$.*

Proof. We will demonstrate this proof in three parts. Let us first show that it is indeed possible to place N queens on the board such that it is a solution. If N and 6 are co-prime, then $1, 2$ and 3 are also co-prime to N. Let σ be the permutation that sends i to $2i \pmod N$. As mentioned before, $\sigma + x$ and $\sigma - x$ are also permutations. In fact, σ sends i to $2i \pmod N$, $\sigma + x$ sends i to $3i \pmod N$ and $\sigma - x$ sends i to $i \pmod N$. We have the following three boards for example with $N = 7$ respectively.

$2i \mod N$ $\qquad$ $3i \mod N$ $\qquad$ $i \mod N$



The first figure is indeed a solution, and all three of them are indeed permutations. As to why the first figure is a solution is pretty simple. It is clear why queens do not attack each other vertically and horizontally. As for diagonally, we know that for both "/" and "\" diagonals, queens that would attack each other would have the same sum $i + j \pmod N$ and difference $i - j \pmod N$ respectively, with i, j denoting the coordinates $0, \ldots, N - 1$. For $2i$, none of the sums $\pmod N$ are equal to each other and none of the differences $\pmod N$ are equal to each other. For instance if we were to have queens on positions $(1, 5)$ and $(4, 1)$, they would attack each other as $1 - 5 \pmod N \equiv 4 - 1 \pmod N$ with $N = 7$.

Next we show that if σ is a permutation of $(0, \ldots, N - 1)$ such that $\sigma + x$ is also a permutation, then N is odd. The permutation $\sigma + x$ can be seen as a single permutation but also as a sum of two permutations, namely the permutation σ and the identity

permutation x. Let us look at both. If $\sigma + x$ is a sum of two permutations, we have

$$\sum_{i=0}^{N-1}(\sigma(i)+i) \pmod N = 2\sum_{i=0}^{N-1} i \pmod N = N(N-1) \pmod N = 0 \pmod N. \tag{1}$$

Note that since σ and x are both permutations of $(0,\ldots,N-1)$, they necessarily have the same sum $\pmod N$, which is why we have twice the sum $\sum_{i=0}^{N-1} i \pmod N$. Now if $\sigma + x$ is a single permutation, we have

$$\sum_{i=0}^{N-1}(\sigma(i)+i) \pmod N = \sum_{i=0}^{N-1} i \pmod N = \frac{N(N-1)}{2} \pmod N = \begin{cases} 0 \pmod N & Nodd \\ \frac{N}{2} \pmod N & Neven \end{cases}.$$

In order to have $\sum_{i=0}^{N-1}(\sigma(i)+i) \pmod N = \sum_{i=0}^{N-1}(\sigma(i)+i) \pmod N$, which we obviously want, we have than N is odd.

Finally, if σ is a permutation of $(0,\ldots,N-1)$ such that $\sigma + x$ and $\sigma - x$ are also permutations, then N cannot be divided by 3. We can again look at $\sigma + x$ and $\sigma - x$ as single permutations but also as sums of two permutations, again permutation σ and identity permutation x. We consider $\sigma + x$ and $\sigma - x$ to be two single permutations and we get

$$\sum_{i=0}^{N-1}(\sigma(i)+i)^2+\sum_{i=0}^{N-1}(\sigma(i)-i)^2 \pmod N = \sum_{i=0}^{N-1} 2\sigma(i)^2+i^2 \pmod N = 4\sum_{i=0}^{N-1} i \pmod N.$$

Note than again since σ and x are both permutations of $(0,\ldots,N-1)$, we have four times the same sum. We now consider $\sigma + x$ and $\sigma - x$ to be two sums of two permutations and we get

$$\sum_{i=0}^{N-1}(\sigma(i)+i)^2 + \sum_{i=0}^{N-1}(\sigma(i)-i)^2 \pmod N = 2\sum_{i=0}^{N-1} i \pmod N.$$

From these two equalities, we get

$$4\sum_{i=0}^{N-1} i \pmod N = 2\sum_{i=0}^{N-1} i \pmod N,$$

which implies that

$$2\sum_{i=0}^{N-1} i \pmod N = \sum_{i=0}^{N-1} i \pmod N.$$

From (1), we get that

$$2\sum_{i=0}^{N-1} i \pmod N = 0 \pmod N.$$

Finally, this implies that

$$0 \pmod N = 2\sum_{i=0}^{N-1} i \pmod N = \frac{2N^3-3N+N}{3} \pmod N = \frac{N}{3}(2N^2-3N+1) \pmod N.$$

If N were divisible by 3, $\frac{N}{3}$ would become a positive integer but $2N^2-3N+1$ would not be divisible by 3. Hence the term $\pmod N$ would not equal 0.

In conclusion, N cannot be even, hence N cannot be divided by 2 and N can also not be divided by 3, hence we have $gcd(N,6) = 1$. □

4.1 Python-code

The following code is based on the permutation code we introduced at the beginning. We simply added the conditions needed for it to be considered as the N Queens puzzle on a toroidal board. Just like the permutation code, the following code works with the permutation method to find all solutions on an $N \times N$ board with N queens. The input is again the variable N, which gives us our $N \times N$ board size and the number of Queens, which will be placed on the board. The output will again consist of the number of total solutions for specific N and it will provide us with a simple illustration of all the distinct solutions.
Program with simple explanations in form of comments (continued from code in Section 2.1.3):

```
from itertools import permutations
# board size and number of Queens
N = 7
# list [0, 1, 2, ..., N-1]
l = list(range(N))
# list of all the permutations of l
perms = permutations(l)
# list of N lists containing N zeros
board = [[0]*N for n in range(N)]
# setboard() will print an NxN board
def setboard():
    for r in range(N):
        print(board[r])
# placequeen(x, y) check if a queen can be placed at (x, y).
# Thus no queen is allowed to placed on the attack field of another
# or on the same field as another
# returns Ture if that is the case, False otherwise
def placequeen(x,y):
    if board[y][x] == 0:
        for k in range(N):
            board[y][k] = '_'
            board[k][x] = '_'
            board[y][x] = 'Q'
            # Attack pattern: diagonalBottomRight
            if y+k <= N-1 and x+k <= N-1:
                board[y+k][x+k] = '_'
                w = y + k
                z = x + k
                for i in range(N):
                    if w == N -1 and z + i < N-1:
                        board[0+i][z+1+i] = '_'
                    if z == N-1 and w + i < N-1:
                        board[w +1 +i][0+i] = '_'
```

```
            # Attack pattern: diagonalTopRight
            if y-k >= 0 and x+k <= N-1:
                board[y-k][x+k] = '_'
                w = y - k
                z = x + k
                for i in range(N):
                    if w == 0 and z + i < N-1:
                        board[N-1-i][z+1+i] = '_'
                    if z == N-1 and w - i > 0:
                        board[w -1 -i][0+i] = '_'
            # Attack pattern: diagonalBottomLeft
            if y+k <= N-1 and x-k >= 0:
                board[y+k][x-k] = '_'
                w = y + k
                z = x - k
                for i in range(N):
                    if w == N -1 and z - i > 0:
                        board[0+i][z-1-i] = '_'
                    if z == 0 and w + i < N-1:
                        board[w +1 +i][N-1-i] = '_'
            # Attack pattern: diagonalTopLeft
            if y-k >= 0 and x-k >= 0:
                board[y-k][x-k] = '_'
                w = y - k
                z = x - k
                for i in range(N):
                    if w == 0 and z - i > 0:
                        board[N-1-i][z-1-i] = '_'
                    if z == 0 and w - i > 0:
                        board[w -1 -i][N-1-i] = '_'
        return True
    else:
        return False


# counting the solutions
count = 0
# We use the permutation method
for p in perms:
    if all(placequeen(p[i],i) == True for i in range(N)):
        count += 1
        print('solution',count,':')
        setboard()
        print(' ')
    board = [[0] * N for n in range(N)]
```

4.2 Explanation of the code above

The code works exactly the same as the permutation code. The only difference occurs at the placequeen() function. That is, because in toroidal chess the queen covers more fields as already mentioned earlier. Here, we added some if statements that take the fact into account that the left and right as well as the bottom and top side of the board are connected with each other.

Example of placequeen()

```
placequeen(1,0)
True
setboard()
['_', 'Q', '_', '_', '_', '_', '_']
['_', '_', '_', 0, 0, 0, 0]
[0, '_', 0, '_', 0, 0, '_']
[0, '_', 0, 0, '_', '_', 0]
[0, '_', 0, 0, '_', '_', 0]
[0, '_', 0, '_', 0, 0, '_']
['_', '_', '_', 0, 0, 0, 0]
placequeen(2,6)
False
```

Output example

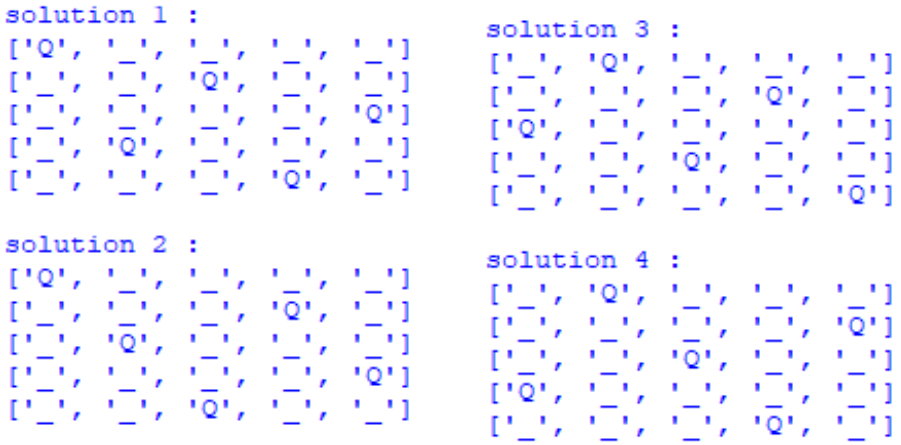
```
solution 7 :
['_', 'Q', '_', '_', '_', '_', '_']
['_', '_', '_', '_', '_', 'Q', '_']
['_', '_', 'Q', '_', '_', '_', '_']
['_', '_', '_', '_', '_', '_', 'Q']
['_', '_', '_', 'Q', '_', '_', '_']
['Q', '_', '_', '_', '_', '_', '_']
['_', '_', '_', '_', 'Q', '_', '_']
```

4.3 Number of Solutions on a Torus

The following table [4, Sequence A000170 and Sequence A051906] shows us the number of solutions on a toroidal chess board in comparison to the number of solutions on a normal chess board.

Number of Queens N	Total Solutions on Normal Board	Torus Solutions
1	1	1
2	0	0
3	0	0
4	2	0
5	10	10
6	4	0
7	40	28
8	92	0
9	352	0
10	724	0
11	2680	88
12	14200	0
13	73712	4524
14	365596	0
15	2279184	0
16	14772512	0
17	95815104	140692
18	666090624	0
19	4968057848	820496
20	39029188884	0
21	314666222712	0
22	2691008701644	0
23	24233937684440	128850048
24	227514171973736	0
25	2207893435808352	1957725000
26	22317699616364044	0
27	234907967154122528	0

For the solutions on a torus, there are always two solutions that are linked together for a certain N. Consider the following examples.



Recall in Theorem 4.4, we talked about the permutations $\sigma, \sigma + x$ and $\sigma - x$, where x is the identity permutation. Now if solution 1 is the permutation σ, then solution 2 is the permutation $\sigma + x$ and similarly if solution 2 is the permutation σ. then solution 1 is the permutation $\sigma - x$. We observe also exactly the same approach for solution 3 and 4.

5 Conclusion

There exist many different approaches to finding solutions for the N queens puzzle. Throughout this project, we have introduced a few, namely the backtracking and the permutation method. There probably are many more. We have come to the conclusion that even if it isn't the most efficient method, that the backtracking method is the go-to method. What we mean by that is that it is way simpler than the permutation method and most people are already familiar with that method, since it is often even used subconsciously in popular games like Solitaire or crosswords for example.
Another reason why we think that the backtracking method has the edge over the permutation method is that it doesn't fix the number of queens that will be placed unlike the permutation method. For example, if we were to use the permutation method for the $N + k$ queens puzzle it would be impossible to handle the already fixed solutions and add k holes and queens to it.

For future work, the concept of $3D$ chess is an interesting research area for the N queens problem. This would be a $N \times N \times N$ board where N boards are stacked on each other and queens can move up and down in addition to the standard movements.

References

[1] Sergio Lopez, Python in Plain English, 2021, URL: https://python.plainenglish.io/coding-the-8-queens-problem-in-python-d168f8df844b

[2] URL: http://www.npluskqueens.info Programs - fourth link

[3] URL: http://www.npluskqueens.info Solutions - Solution count

[4] N. J. A. Sloane, The On-Line Encyclopedia of Integer Sequences (OEIS) URL: https://oeis.org

[5] Wikipedia contributors. (2022, May 16). Cylinder chess. Wikipedia. https://en.wikipedia.org/wiki/Cylinder_chess

[6] Wikipedia contributors. (2022, April 14). Torus. Wikipedia. https://en.wikipedia.org/wiki/Torus

[7] Wikipedia contributors. (2022, April 30). Eight queens puzzle. Wikipedia. https://en.wikipedia.org/wiki/Eight_queens_puzzle

[8] R. Douglas Chatham, Gerd H. Fricke, R. Duane Skaggs, The Queens Separation Problem, 2004 URL: http://www.npluskqueens.info/papers.html

[9] Schlude, Konrad, and Ernst Specker. "Zum Problem der Damen auf dem Torus." Elemente der Mathematik 66.3 (2011): 89-97.