Python implementation of the N-queens problem

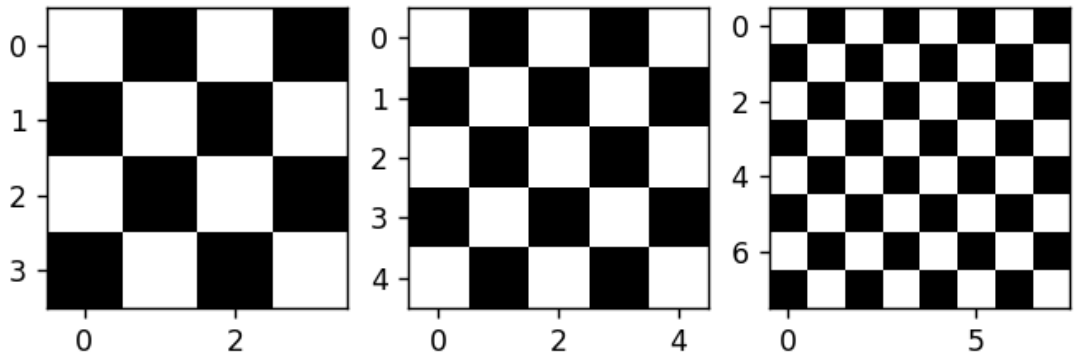
Introduction

This memo is all about the explanation of the very basis of the n queens problem and the reasoning behind the ulteriorly implemented algorithm in python. The main goal is to understand how a recursive algorithm is able to navigate throughout all the different branches that conform the tree this problem represents, and how it makes sure it finds all posible solutions within the given set of conditions that we will be working with.

The N queens problem

Imagine we have a NxN chessboard, where each tile is either black or white and none of these squared tiles are in direct contact with any other tile that is colored the same way*.

Take these 4x4,5x5 and 8x8 boards as reference examples.



> *note that we do not care about the color of the starting tile, as it does not affect the ending result

Given this setup, our goal is to place N queens in each one of them, so that none of them is threatening any tile containing any of the remaining ones.

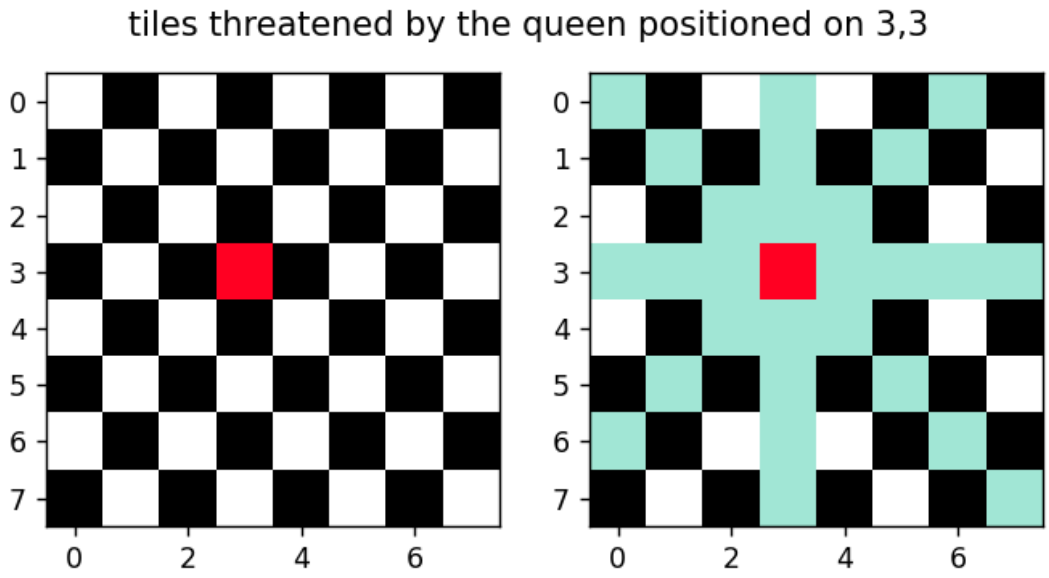
When we reference queens, it's important to understand the possible movements that a queen is able to perform and therefore put the respectively reference tiled, into a threatened state.

The rules are quite simple:

- A queen can move anywhere in it's row
- A queen can move anywhere in it's column
- A queen can move anywhere in diagonal, thus meaning staying in the same color as the tile corresponding to the starting point and having a vertical displacement as large as it's horizontal displacement, in all 4 directions it's able to perform

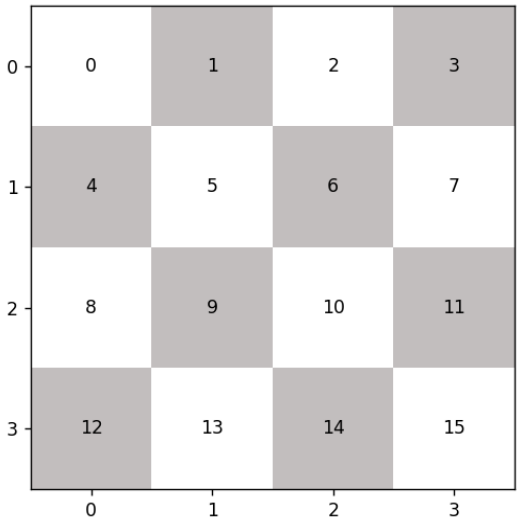
Although this might be a little of an overexplanation, these concepts are rather relevant when it comes to formalizing what our possibilities look like when it comes to finding the queen-threatened tiles.

For reference:

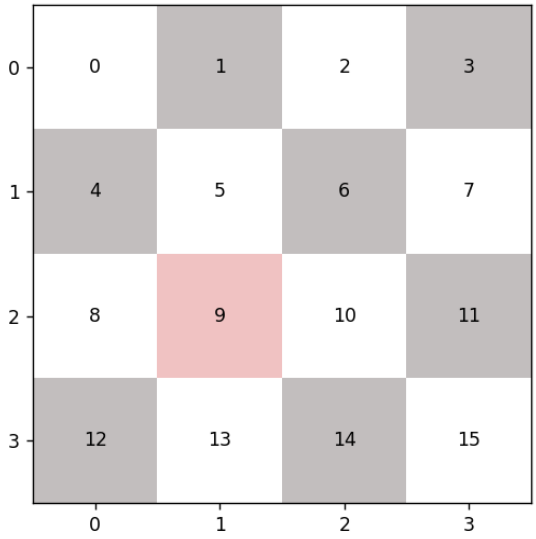


The mapping of the board

Let´s now assign a number to every tile on the board. Not an arbitrary number, of course, but a number that does simultaneously resemble it´s corresponding row and column.



Note that any number whose module on base 4 (a%4) is 0 will correspond to the first column, and any number whose whole number division result (a//4) is 0, is referencing some tile on the first row. This same logic can be applied to any number in this setup, since they all satisfy these two properties needed in order to make sense of it. Take for example 9, whose module is 1 (4*2 +**1**), and once divided by 4 it gives us 2 (4***2**+1), therefore corresponding to row 2 column 1.



Thus, given the set of all posible tiles $b = \{0, 1...N^2 - 1\}$ for any given N, we can reference the row any tile k belongs to by taking all the numbers such that $n//N = k//N$

```
{n for n in b if k//N == n//N} #all numbers n within the same row as k
```

```
{n for n in b if n%N == k%N} # all numbers n within the same column as k
```

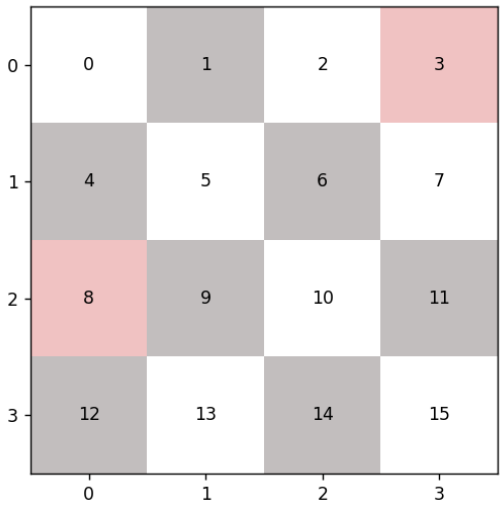
And applying a similar reasoning we get to how we can obtain the numbers contained in the column our number k belongs to, this being all numbers n such that $(n)modN = (k)modN$

The notation b for tiles not yet occupied will be used along the code and throughout explanations, and k will be our way to reference the specific tile we are trying to take.

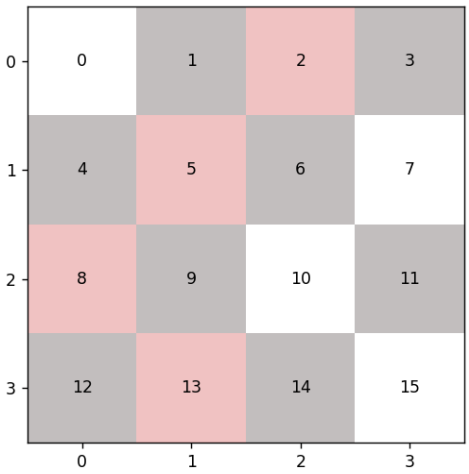
The diagonals

So far both rows and cols have been rather easy to understand but for diagonals we get into something a little more complicated. Note that any element that falls in diagonal respective to our element k satisfies the formula $0 < k + rN + r < N^2$, where r is an integer. This doesn't mean that if you manage to satisfy this you obtained a tile placed diagonally. take for example the number 8 in a 4x4 board.

The number 3 satisfies such formula yet it is not in diagonal. $3 = 8 - 1 \cdot 4 - 1$.



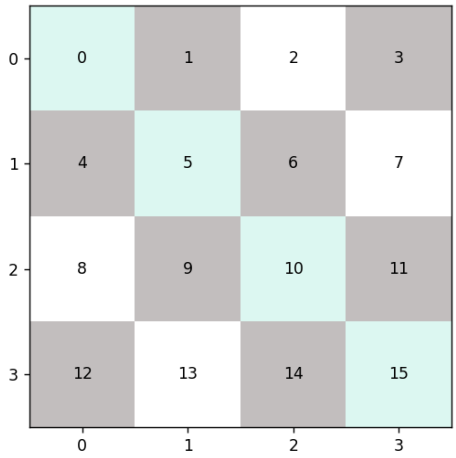
But the idea of moving to some other row and some other column following that formula is still what were looking for.



Notice how for all numbers, we can find some r so that $k' = k + rN + r$

The first diagonal

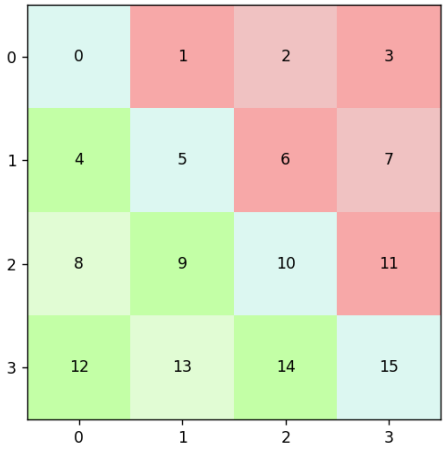
First it's important to separate our board (not the 4x4 but any given board) into three separate parts, those being the 'main' diagonal, that goes from the first tile 0 down to the tile $N^2 - 1$, satisfying that for any element k, $(k)modN = k//N$:



All the numbers contained here satisfy our requirements, but it's even better than just that. For any r and k given (as far as k belongs to our main diagonal), any element k' will still belong to the main diagonal (note that $0 < k' < N^2$), which means we do no longer concern about tiles that may or may not be in diagonal respective to our tile k, because we are completely sure they all will be.

Secondary areas

When we reference the two secondary areas, we´re talking about the two remaining spaces aside from the tiles on the first diagonal. For reference:



Now it's important to understand the main issue here: borders.

The green area

This area represents all numbers such that $(k) mod N < k//N$

If we were to take any number in the first column, green area, and try to generate some k' above them, knowing we´re working with the topleft-botright diagonal, we´ll find ourselves in some trouble. Just to give some examples, take the previous $3 = 8 - 1 \cdot 4 - 1$, same trouble appears when trying to do the same thing with any negative r value: the result is not where we want it to be because there´s no tile there. The same problem appears when working with last column, red area, taking again the same example: $8 = 3 + 1 \cdot 4 + 1$. Therefore we need to find some way of restricting which r values fit our model. For the green area, we can go down as much as we want, since it will, worst case scenario, result in some $k' > N^2$, which we will easily deny from belonging into our set of tiles.

Trouble comes with going upwards, because that's how we might encounter some border by moving to the left, which translates into some $r < 0$. Our way to solve this is to take a look at the module of our tile, which will tell us how may tiles is it away from crashing against a wall if it was to go left. Therefore, for our green area, $-(k)modN \leq r < N$. Take for example 14: since its two tiles away from the left border, we know that our lowest possible r value is -2.

The red area

This area represents all numbers such that $(k)modN > k//N$

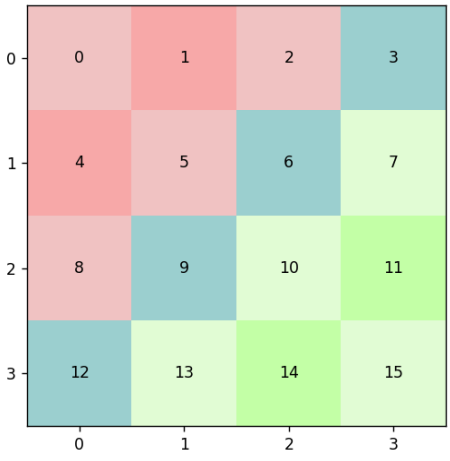
As you might have already guessed, we are now encountering the same problem with borders in the red area. but in this case, It´s just the other way around. We can go upwards as much as we want cause again, simlarly to the green area, in the red area our worst case scenario is some

$k' < 0$, which will be discarded. As for how large can r get, it's all about the column, because our last column is directly next to a border. Thus, r can reach it's highest value when it's on its furthest position from the border. Now we can conclude that $-N \leq r < N - kmodN$. Take for example 1, whose module is also 1 in our 4x4 board. according to the previous statement, our value r reaches it's maximum value at 2, the last integer lesser than N-1 (3).

Let's now see how we can translate this into some code:

The second diagonal

It is rather intuitive to imagine what this one looks like, but the formalization of it is that this set of tiles contains any tile k such that $(k)modN = (N - k//N - 1)$, which might sound a little confusing at first, but all it´s saying is that for any tile you move downwards, you must move left as well. It's also important to note that now $k' = k + rN - r$, contrary to our first diagonal whose formula for k' was $k + rN + r$



Again, we separated our board into 3 parts, those being the main diagonal and the remaining two areas.

As for the main Diagonal, our r value will depend on k. To put it simply, we again don't want to face any borders so we can restrict it according to its row. Therefore in the first row, $0 < r < N$, and for the

last one, $-N < r < 0$, because we don't want to go any further than the length of the actual board. Therefore, we can simplify it into this expression: $-(k//N) \leq r < N - k//N$

.

The red area

This area represents all numbers such that $(k)modN < (N - k//N - 1)$.

Here we are able to move upwards as much as we wish, but the amount of tiles we are able to navigate down (and left) is limited by the column we're in, therefore: $-N \leq r < (k)modN + 1$, because for $(k)modN = 0$ we still want to be able to take 0 as a value of r. This makes sure that if we get too close to the border, no left movements will be performed.

The green area

This final area represents all numbers such that $(k)modN > (N - k//N - 1)$.

In this situation, we can freely move downwards without worrying about crashing, therefore our upper bound for r will be just N. As for the lower bound, we need to again take a look at the column we're in, thus giving us the expression $-N + (k)modN + 1 \leq r < N$, where we use the lower bound to reference that the more to the right our tile is placed, the less we can move right upwards, because we are placed closer to the right border.

Note that…

Until now we have always wrote r as some integer in an interval defined as $A \leq r < B$, this is intended to make it something that works as the for loop works in python, were we give a lower bound, starting point for our value r, and an upper bound, last value of our loop which r never actually reaches, therefor making the < operator fit our explanation better.

Threats

Now that we have explained how we can obtain all tiles threatened by a queen in the tile k, let¡s see how we can implement these ideas in Python.

```
def threats(k,b,N):
    #b is our set of yet to be threatened tiles
    #k is the tile our queen has been placed
    #N is the size of the board (NxN)

    def valid(n,N):
        return 0<=n<(N**2) #returning true or false, makes sure n is within the board

    def maindiag(k,N):

        #main top left- bottom right diagonal
        if k%N == k//N:
            return -N,N

        #red area
        elif k%N > k//N:
            return -N,N-k%N

        #green area
        else:
            return -(k%N),N
```

```
    def secdiag(k,N):
        #main top right- bottom left diagonal
        if k%N == N-k//N-1:
            return -(k//N),N-k//N

        #green area
        elif k%N > N-k//N-1:
            return -N+k%N+1,N

        #red area
        else:
            return -N,k%N+1

    line = {n for n in b if k//N == n//N}

    row = {n for n in b if n%N == k%N}

    diags1 = {k+r+r*N for r in range(*maindiag(k, N)) if valid(k+r+r*N, N)}
    # we create a for loop with our desired bounds as inputs, and make sure the result is still in the board

    diags2 = {k-r+r*N for r in range(*secdiag(k, N)) if valid(k-r+r*N, N)}
    #same as the previous line, but for second diagonal.

    return b-(line|row|diags1|diags2|{k})#returns all previously unthreatened tiles
                                        #excluding the ones this new queen k is now threatening.
```

The algorithm:

Now that we know how to precisely determine what our possibilities are after placing a queen, let's see what can we now do in order to find a way to place them all.

At the very beginning of this memo, we talked about how this problem essentially represents a tree of possibilities, and that is what we will be working with now: paths and branches.

Imagine a simple board, a 3x3 as a start (although there is not any solution for N=3, we can still see what we are looking forward to do).

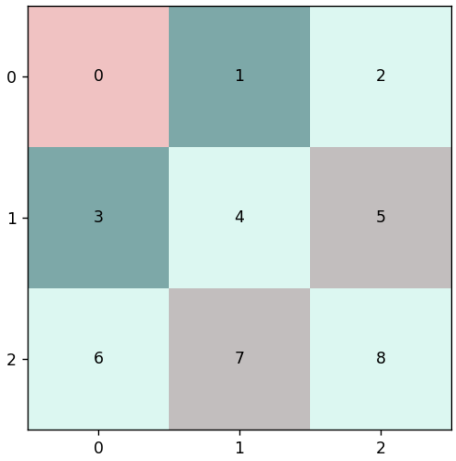
For a 3x3 board, we have $b = \{0, 1, 2..., 9\}$

We will be trying every single possible scenario, so let´s start by trying with $k = 0$:

We initiate a variable denominated positions, consisting of a list of the tiles taken so far.

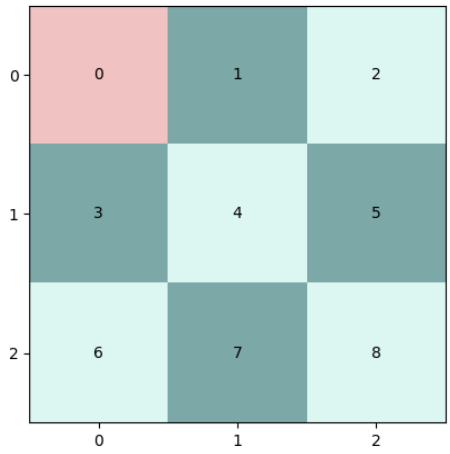
```
b = {n for n in range(N**2)}
k=0
positions = [] #
solutions = []
b = threats(0,b,N) #will redefine b as {5,7}, the now only possible tiles to take
```

Let's picture that:



Now we find ourselves in a situation where only two possible paths can be tajen: either placing our new queen in 5, or in 7.

Now positions = [0] and solutions is still empty. We continue along this path, taking tile 5 next, making positions now equal to [0,5].



All tiles have now been either threatened or occupied now, therefore we must check if we have found a solution, by checking if the number of occupied tiles is as large as our N (3).

Since positions = [0,5] has 2 elements, we conclude that we have not found a solution, therefore taking us to the most recent step where a choice could be made: positions = [0]. Yet again we try to take the next possible tile: 7, and we end up in the same scenario as before, where positions is a two element list.

Now that we have checked all possible scenarios for taking 0 as a starting spot, we must now try to find a solution taking 1 as our starting point, and repeat the same process.

Summarizing, our goal is to rey to find a solution via placing queens every time it’s possible to, and if we are unable to do so, go back to our last choice scenario and check all possible paths.

```
def queens(b,N,sols,currentposs):

    for i in b: #for every possible choice in our current scenario
        tmp = currentposs.copy() # we 'save' our last choice scenario.
        tmp.append(i) # last tile occupied is added to our potential solution
        j = threats(i,b,N) # we redefine the now possible tiles to be taken

        if len(tmp) == N: #this would mean wwe have found a solution
            sols.append(tmp)

        if len(j) == 0: # if we have no tiles left but we havent found a solution, we simply keep trying
            continue


        queens(j,N,sols,tmp) #we now try every possible decision now that we havr placed
                             # a new quueen, therefore calling the same function, passing
                             # tmp as the last known set of decisions made so far

    return sols
```

Conclusions

Although the code is rather simple, it’s the reasoning behind it that might be confusing. To put it simply, every time we find a dead end we try again, and we can do that passing the last choice made to every posible scenario. This idea can be applied to many different problems that require trying out different possibilities and choices, as well as pathings and step by step solutions.

Note that this algorithm is not aimed towards efficiency, since there are many changes we could make due to logical reasoning, such as narrowing down the starting trial tiles to only the ones on the first row, or converting every list into a set to avoid repeated solutions. However, this way we can get a full understanding on how this idea can be applied to scenarios where we might not know what cases are simply avoidable, or not taken into account.