Backtracking

Introduction

- A **backtracking** algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.
- The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.
- The term **backtracking** suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.
- This approach is used to solve problems that have multiple solutions.
- Backtracking is thus a form of recursion.
- We begin by choosing an option and backtrack from it, if we reach a state where we conclude that this specified option does not give the required solution.
- We repeat these steps by going across each available option until we get the desired solution.

There are three types of problems in backtracking:

- **Decision Problem:** In this, we search for a feasible solution
- **Optimization Problem:** In this, we search for the best solution
- **Enumeration Problem:** In this, we find all feasible solutions

Difference between Recursion and Backtracking

In recursion, the function calls itself until it reaches a base case. In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

Below is an example of finding all possible order of arrangements of a given set of letters. When we choose a pair we apply backtracking to verify if that exact pair has already been created or not. If not already created, the pair is added to the answer list, else it is ignored.

```
def permute(list, s):
    if list == 1:
        return s
    else:
        return [y+x for y in permute(1, s) for x in permute(list - 1, s)]

print(permute(1, ["a","b","c"]))
print(permute(2, ["a","b","c"]))
```

When the above code is executed, it produces the following result −

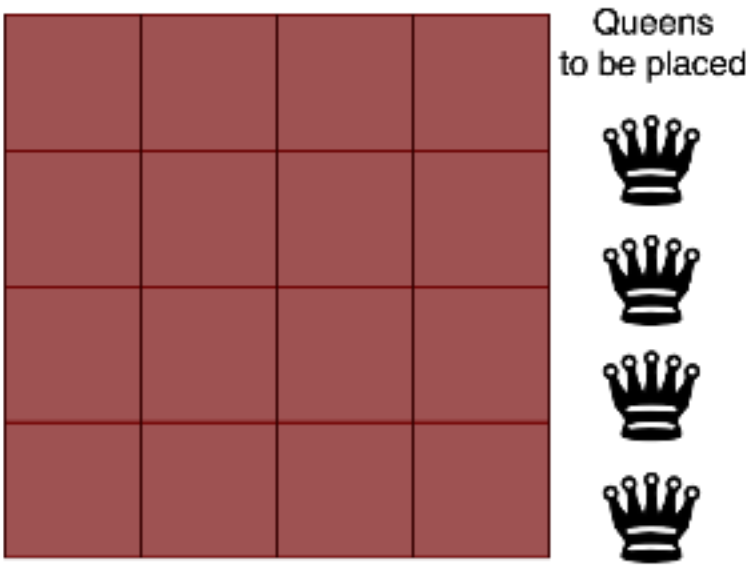
```
['a', 'b', 'c']
['aa', 'ab', 'ac', 'ba', 'bb', 'bc', 'ca', 'cb', 'cc']
```

Problem Statement: N-Queen

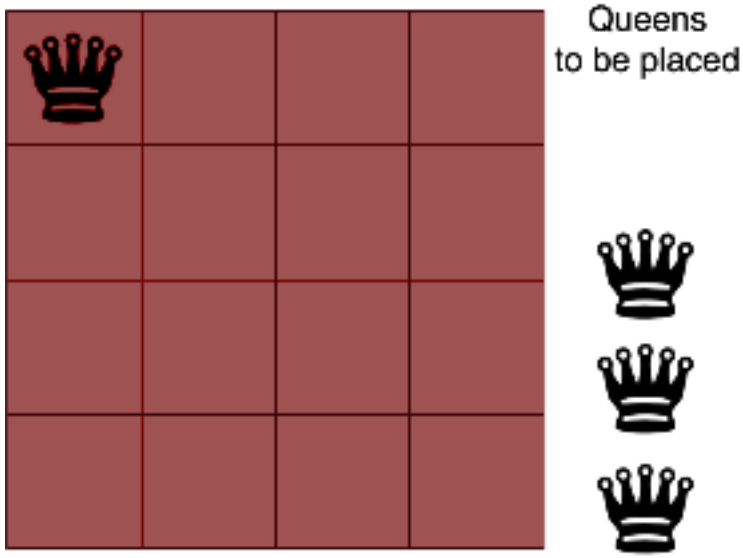
One of the most common examples of the backtracking is to arrange N queens on an NxN chessboard such that no queen can strike down any other queen. A queen can attack horizontally, vertically, or diagonally.

The solution to this problem is also attempted using Backtracking.

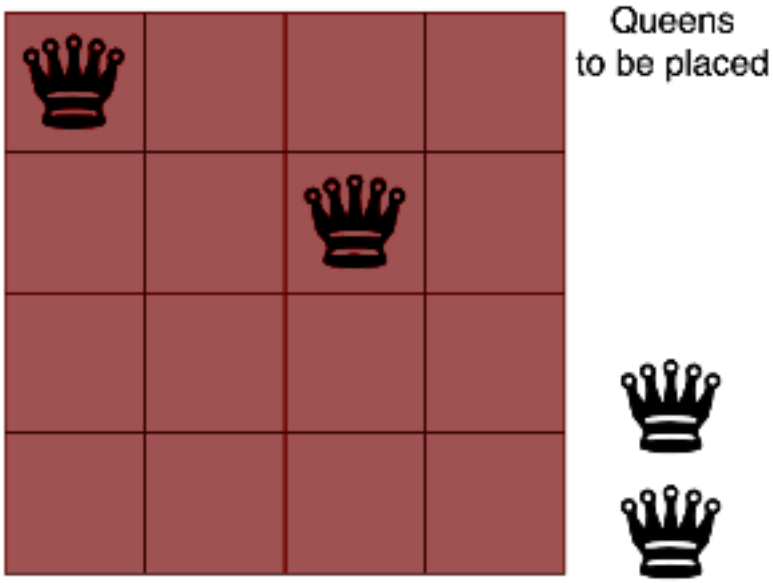
- We first place the first queen anywhere arbitrarily and then place the next queen in any of the safe places.
- We continue this process until the number of unplaced queens becomes zero (a solution is found) or no safe place is left.
- If no safe place is left, then we change the position of the previously placed queen.

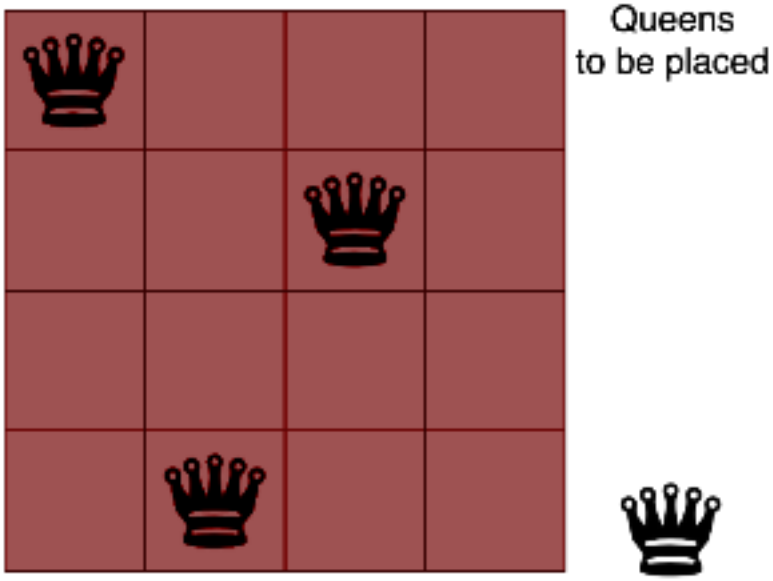


- The above picture shows an NxN chessboard and we have to place N queens on it. So, we will start by placing the first queen.

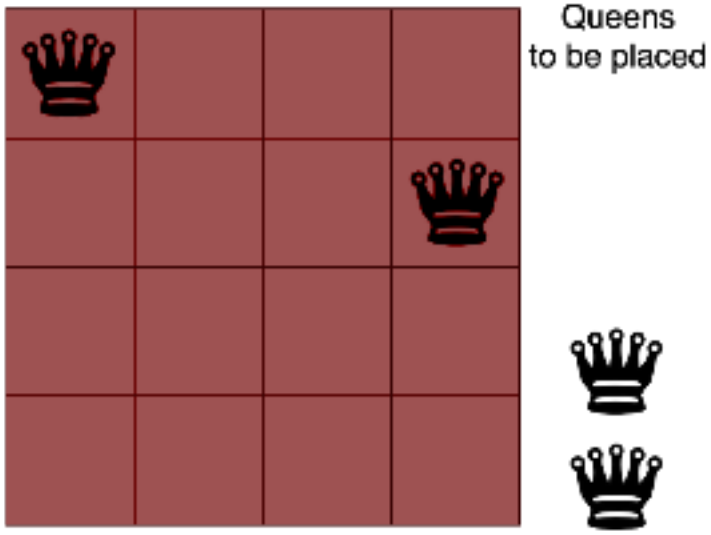


- Now, the second step is to place the second queen in a safe position and then the third queen.

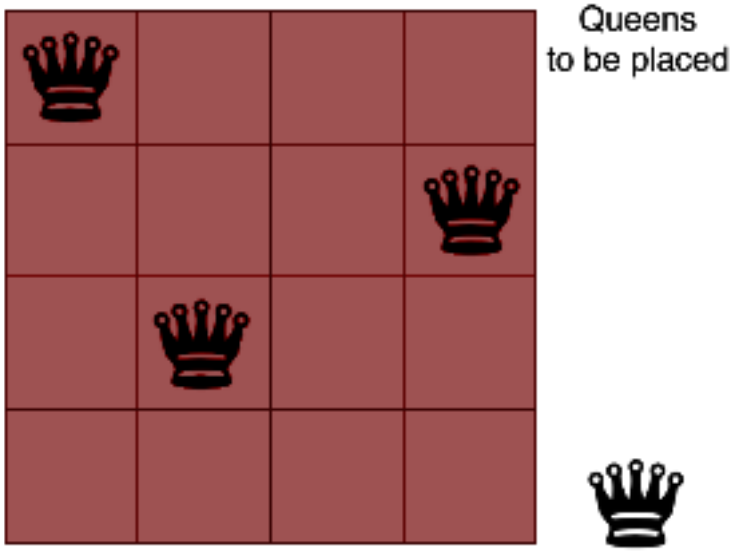




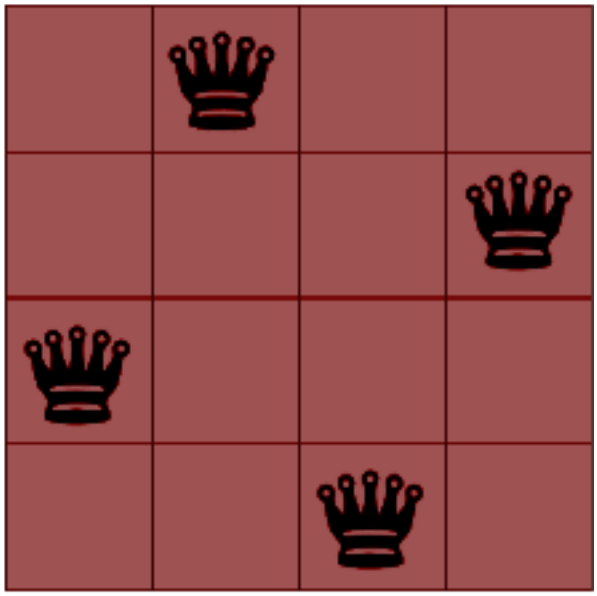
- Now, you can see that there is no safe place where we can put the last queen. So, we will just change the position of the previous queen. And this is backtracking.
- Also, there is no other position where we can place the third queen so we will go back one more step and change the position of the second queen.



- And now we will place the third queen again in a safe position until we find a solution.



- We will continue this process and finally, we will get the solution as shown below.



As now you have understood backtracking, let us now code the above problem of placing N queens on an NxN chessboard using the backtracking method.

```
#Number of queens
print ("Enter the number of queens")
N = int(input())

#NxN matrix with all elements 0
board = [[0]*N for _ in range(N)]

def check_possible(i, j):
    #checking if there is a queen in row or column
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonals
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
```

```
def N_queen(n):
    #if n is 0, solution found
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            '''checking if we can place a queen here or not
            queen will not be placed if the place is being attacked
            or already occupied'''
            if (not(check_possible(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                #recursion
                #check if we can put a queen in this arrangement
                if N_queen(n-1)==True:
                    return True
                board[i][j] = 0

    return False

N_queen(N)
for i in board:
    print (i)
```

Explanation of the code

- `check_possible(int i,int j)` → This is a function to check if the cell (i,j) is under attack by any other queen or not. We are just checking if there is any other queen in the row 'i' or column 'j'. Then we are checking if there is any queen on the diagonal cells of the cell (i,j) or not. Any cell (k,l) will be diagonal to the cell (i,j) if k+l is equal to i+j or k-l is equal to i-j.
- `N_queen` → This is the function where we are implementing the backtracking algorithm.
- `if(n==0)` → If there is no queen left, it means all queens are placed and we have got a solution.

- `if((!check_possible(i,j)) && (board[i][j]!=1))` → We are just checking if the cell is available to place a queen or not. check_possible function will check if the cell is under attack by any other queen and `board[i][j]!=1` is making sure that the cell is vacant. If these conditions are met then we can put a queen in the cell – `board[i][j] = 1`.
- `if(N_queen(n-1)==1)` → Now, we are calling the function again to place the remaining queens and this is where we are doing backtracking. If this function (for placing the remaining queen) is not true, then we are just changing our current move – `board[i][j] = 0` and the loop will place the queen in some other position this time.

Another Example: Rat in A Maze

Go through the given blog to get a deeper understanding of the **Rat in A Maze** Problem:

https://www.codingninjas.com/blog/2020/09/02/backtracking-rat-in-a-maze/