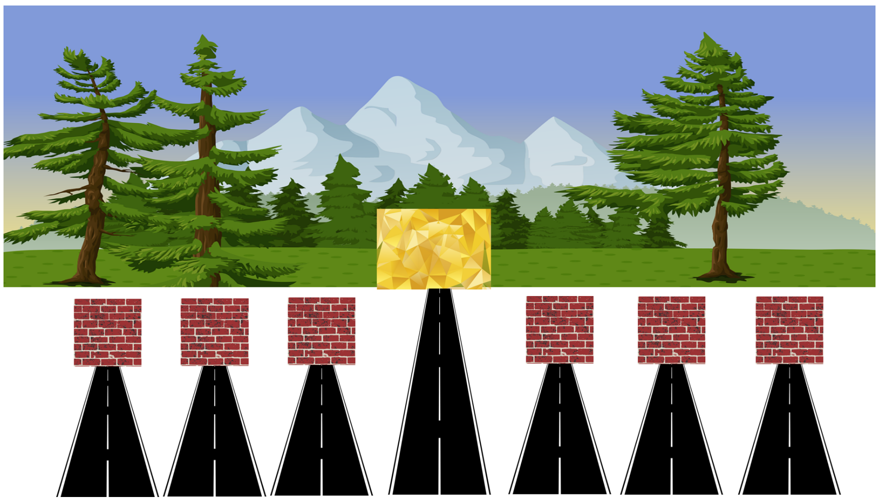
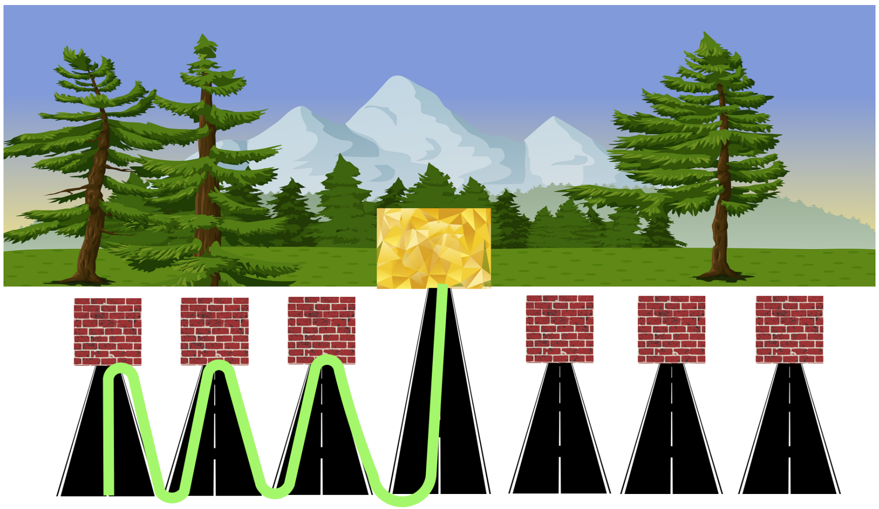
Backtracking | N-Queens Problem

Suppose there is a gold mine somewhere in a jungle and you are standing outside the jungle. There are 10 different paths which are going to the jungle, out of which only one path is going to lead to the mine. What would you do?



There is no option other than checking every path until the mine is found. So, you will start from the first path and if the mine is not found using this path, you will come back to take the second path and so on. This is backtracking, you just backtracked to the origin to take a different path when the mine was not found.



There can be more than one path leading to the mine. In that case, we use backtracking to find just one solution or more solutions depending upon the necessity of the problem.

Think about the problems like finding a path in a maze puzzle, assembling lego pieces, sudoku, etc. In all these problems, backtracking is the natural approach to solve them because all these problems require one thing - if a path is not leading you to the correct solution, come back and choose a different path.

Thus, we start with a sub-solution of a problem (which may or may not lead to the correct solution) and check if we can proceed further with this sub-solution or not. If not, then we just change this sub-solution. So, the steps involved are

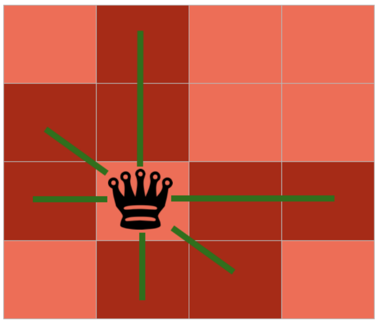
- Start with a sub-solution.
- Check if this sub-solution will lead to a solution or not.
- If not, then change the sub-solution and continue again.

Take note that even tough backtracking solves the problem but yet it doesn't always give us a great running time. For example, you will see factorial running time in many cases with backtracking but yet we can use it to solve problems with small size (like most of the puzzles).

Let's get our hands dirty and use backtracking to solve N-Queens problem.

N Queens Problem

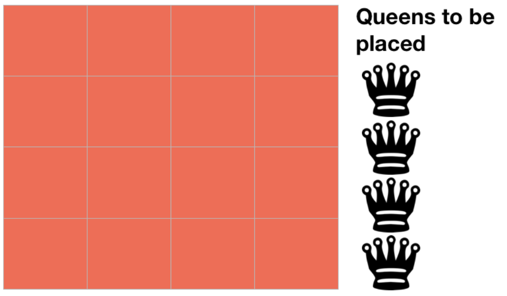
N queens problem is one of the most common examples of backtracking. Our goal is to arrange N queens on an NxN chessboard such that no queen can strike down any other queen. A queen can attack horizontally, vertically, or diagonally.


Cells attacked by the queen

So, we start by placing 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.

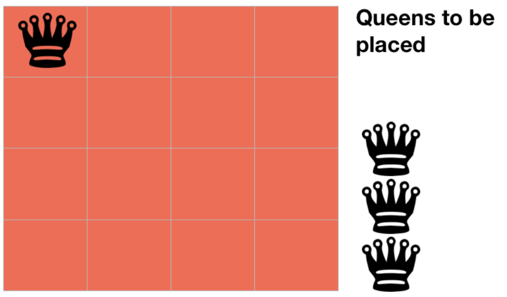
Let's test this algorithm on a 4x4 chessboard.

Using Backtracking to Solve N Queens

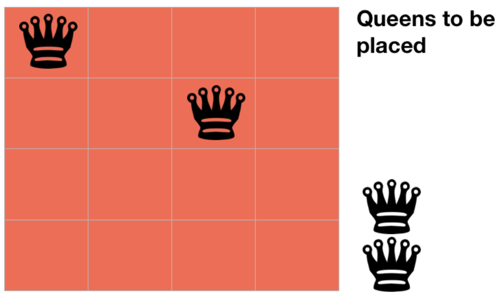


4x4 Chessboard

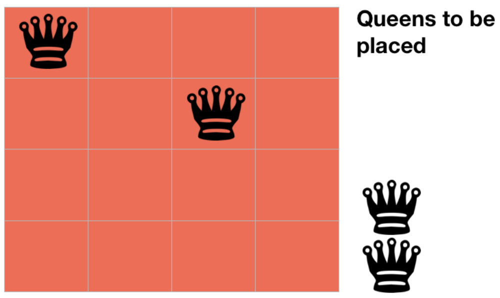
The above picture shows a 4x4 chessboard and we have to place 4 queens on it. So, we will start by placing the first queen in the first row.



Now, the second step is to place the second queen in a safe position. Also, we can't place the queen in the first row, so we will try putting the queen in the second row this time.

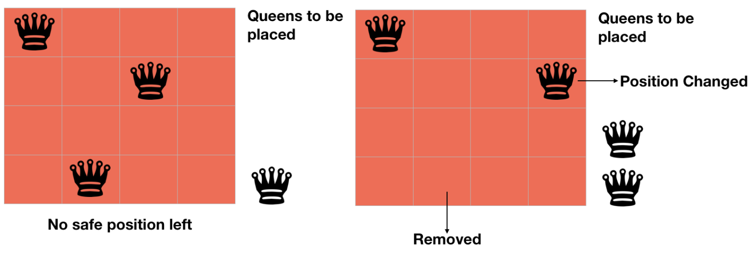


Let's place the third queen in a safe position, somewhere in the third row.

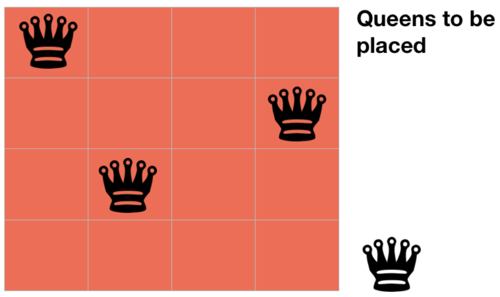


Now, we can see that there is no safe place where we can put the last queen. So, we will just change the position of the previous queen i.e., backtrack and change the previous decision.

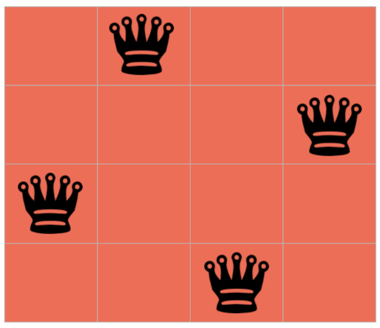
Also, there is no other position where we can place the third queen, so we will go back one more step and change the position of the second queen.

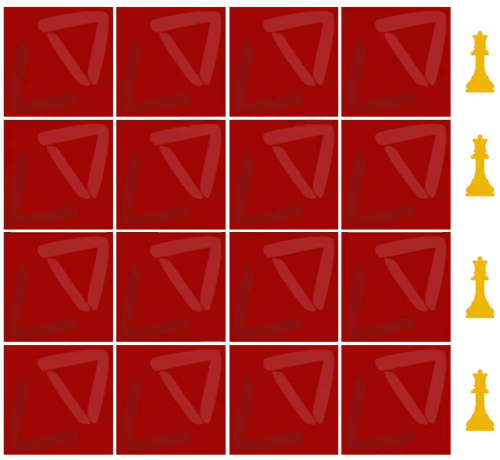


And now we will place the third queen again in a safe position other than the previously placed position in the third row.



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





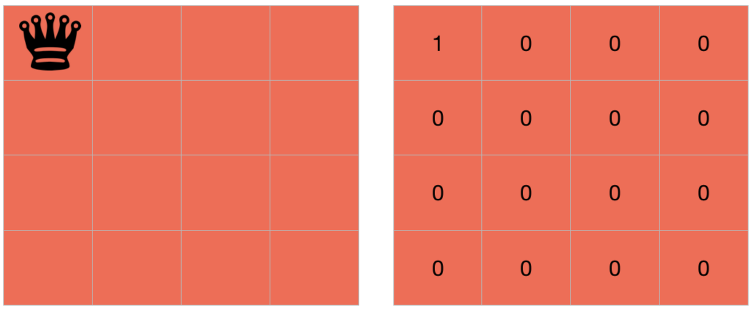
After understanding the backtracking and N queens problem, let's write the code for it.

Code for N Queens

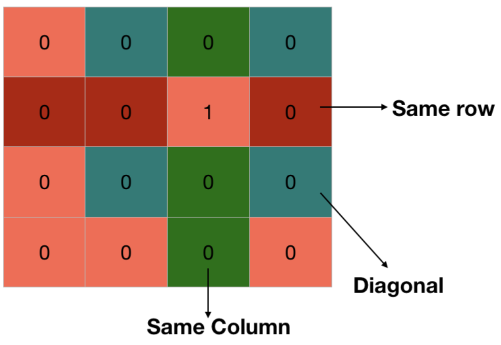
Let's first write a function to check if a place is safe to put a queen or not.

We need to check if a cell (i, j) is under attack or not. For that, we will pass these two in our function along with the chessboard and its size - `IS-ATTACK(i, j, board, N)`.

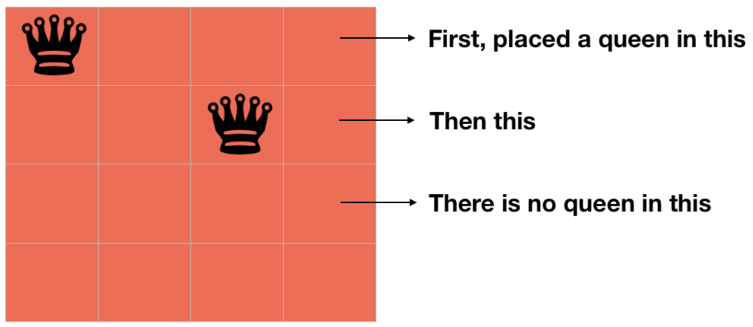
If there is a queen in a cell of the chessboard, then its value will be 1, otherwise, 0.



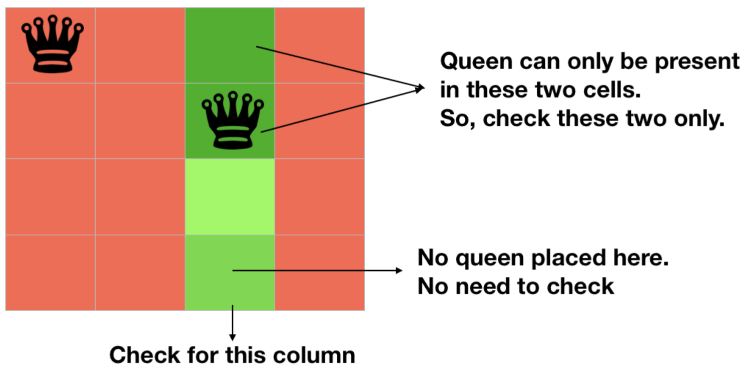
The cell (i,j) will be under attack in three condition - if there is any other queen in row i, if there is any other queen in the column j or if there is any queen in the diagonals.



We are already proceeding row-wise, so we know that all the rows above the current row(i) are filled but not the current row and thus, there is no need to check for row i.



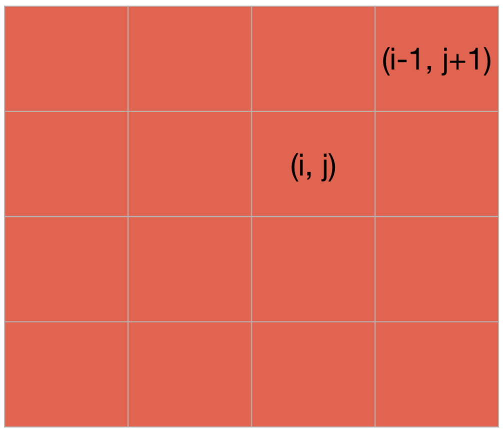
We can check for the column j by changing k from 1 to i-1 in `board[k][j]` because only the rows from 1 to i-1 are filled.


Queen can only be present in these two cells. So, check these two only.
No queen placed here. No need to check
Check for this column

```
for k in 1 to i-1
   if board[k][j]==1
      return TRUE
```

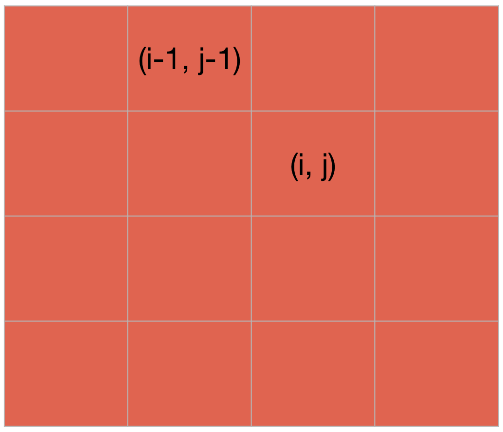
Now, we need to check for the diagonal. We know that all the rows below the row i are empty, so we need to check only for the diagonal elements which above the row i.

If we are on the cell (i, j), then decreasing the value of i and increasing the value of j will make us traverse over the diagonal on the right side, above the row i.



```
k = i-1
l = j+1
while k>=1 and l<=N
   if board[k][l] == 1
      return TRUE
   k=k-1
   l=l+1
```

Also if we reduce both the values of i and j of cell (i, j) by 1, we will traverse over the left diagonal, above the row i.



```
k = i-1
l = j-1
while k>=1 and l>=1
  if board[k][l] == 1
    return TRUE
  k=k-1
  l=l-1
```

At last, we will return false as it will be return true is not returned by the above statements and the cell (i,j) is safe.

We can write the entire code as:

```
IS-ATTACK(i, j, board, N)
  // checking in the column j
  for k in 1 to i-1
    if board[k][j]==1
      return TRUE

  // checking upper right diagonal
  k = i-1
  l = j+1
  while k>=1 and l<=N
    if board[k][l] == 1
      return TRUE
    k=k+1
    l=l+1

  // checking upper left diagonal
  k = i-1
  l = j-1
  while k>=1 and l>=1
    if board[k][l] == 1
      return TRUE
    k=k-1
    l=l-1

  return FALSE
```

Now, let's write the real code involving backtracking to solve the N Queen problem.

Our function will take the row, number of queens, size of the board and the board itself - `N-QUEEN(row, n, N, board)`.

If the number of queens is 0, then we have already placed all the queens.

```
if n==0
  return TRUE
```

Otherwise, we will iterate over each cell of the board in the row passed to the function and for each cell, we will check if we can place the queen in that cell or not. We can't place the queen in a cell if it is under attack.

```
for j in 1 to N
  if !IS-ATTACK(row, j, board, N)
    board[row][j] = 1
```

After placing the queen in the cell, we will check if we are able to place the next queen with this arrangement or not. If not, then we will choose a different position for the current queen.

```
for j in 1 to N
  ...
    if N-QUEEN(row+1, n-1, N, board)
      return TRUE

    board[row][j] = 0
```

`if N-QUEEN(row+1, n-1, N, board)` - We are placing the rest of the queens with the current arrangement. Also, since all the rows up to 'row' are occupied, so we will start from 'row+1'. If this returns true, then we are successful in placing all the queen, if not, then we have to change the position of our current queen. So, we are leaving the current cell `board[row][j] = 0` and then iteration will find another place for the queen and this is backtracking.

Take a note that we have already covered the base case - `if n==0 → return TRUE`. It means when all queens will be placed correctly, then `N-QUEEN(row, 0, N, board)` will be called and this will return true.

At last, if true is not returned, then we didn't find any way, so we will return false.

```
N-QUEEN(row, n, N, board)
...
  return FALSE
```

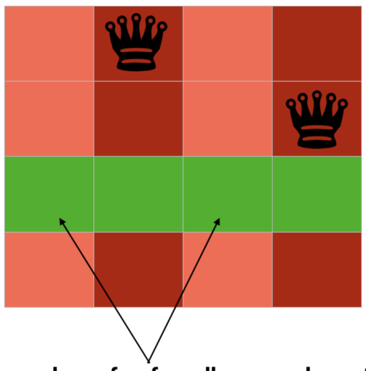
```
N-QUEEN(row, n, N, board)
  if n==0
    return TRUE

  for j in 1 to N
    if !IS-ATTACK(row, j, board, N)
      board[row][j] = 1

      if N-QUEEN(row+1, n-1, N, board)
        return TRUE

      board[row][j] = 0 //backtracking, changing current decision
  return FALSE
```

C **Python** **Java**

```c
#include <stdio.h>

int is_attack(int i, int j, int board[5][5], int N) {
  int k, l;
  // checking for column j
  for(k=1; k<=i-1; k++) {
    if(board[k][j] == 1)
      return 1;
  }

  // checking upper right diagonal
  k = i-1;
  l = j+1;
  while (k>=1 && l<=N) {
    if (board[k][l] == 1)
      return 1;
    k=k+1;
    l=l+1;
  }

  // checking upper left diagonal
  k = i-1;
  l = j-1;
  while (k>=1 && l>=1) {
    if (board[k][l] == 1)
      return 1;
    k=k-1;
    l=l-1;
  }

  return 0;
}

int n_queen(int row, int n, int N, int board[5][5]) {
  if (n==0)
    return 1;

  int j;
  for (j=1; j<=N; j++) {
    if(!is_attack(row, j, board, N)) {
      board[row][j] = 1;

      if (n_queen(row+1, n-1, N, board))
        return 1;

      board[row][j] = 0; //backtracking
    }
  }
  return 0;
}

int main() {
  int board[5][5];
  int i, j;

  for(i=0;i<=4;i++) {
    for(j=0;j<=4;j++)
      board[i][j] = 0;
  }

  n_queen(1, 4, 4, board);

  //printing the matix
  for(i=1;i<=4;i++) {
    for(j=1;j<=4;j++)
      printf("%d\t",board[i][j]);
    printf("\n");
  }
  return 0;
}
```

Analysis of N Queens Problem

The analysis of the code is a little bit tricky. The for loop in the N-QUEEN function is running from 1 to N (N, not n. N is fixed and n is the size of the problem i.e., the number of queens left) but the recursive call of `N-QUEEN(row+1, n-1, N, board)` ($T(n-1)$) is not going to run N times because it will run only for the safe cells. Since we have started by filling up the rows, so there won't be more than n (number of queens left) safe cells in the row in any case.



Maximum number of safe cells = number of queens left

So, this part is going to take $n*T(n-1)$ time.

Also, the for loop is making N calls to the function IS-ATTACK and the function has a $O(N-n)$ worst case running time.

Since $(N-n) \leq N$, therefore, $O(N-n) = O(N)$.

Thus,

$$T(n) = O(N^2) + n*T(n-1)$$

Replacing $T(n-1)$ with $O(N^2) + (n-1)T(n-2)$,

$$T(n) = O(N^2) + n*\left(O(N^2) + (n-1)T(n-2)\right)$$
$$= O(N^2) + nO(N^2) + n(n-1)T(n-2)$$

Replacing $T(n-2)$,

$$T(n) = O(N^2) + nO(N^2) + n(n-1)\left(O(N^2) + (n-2)T(n-3)\right)$$
$$= O(N^2) + nO(N^2) + n(n-1)O(N^2) + n(n-1)(n-2)T(n-3)$$

Similarly,

$$T(n) = O(N^2)\left(1 + n + n(n-1) + n(n-2) + \ldots\right) + n*(n-1)*(n-2)*(n-3)*(n-4)*\ldots*T(0)$$
$$T(n) = O(N^2)\left(O((n-2)!)\right) + n*(n-1)*(n-2)*(n-3)*\ldots*T(0)$$
$$= O(N^2)\left(O((n-2)!)\right) + O(n!)$$

The above expression is dependent upon both the size of the board (N) and the number of queens (n). One can think that the term $O(N^2)\left(O((n-2)!)\right)$ will dominate if N is large enough but this is not going to happen.

Think about placing 1 queen on a 4x4 chessboard. Even if the size of the board (N) is quite greater than the number of queen (n), the algorithm will just find a place for the queen and then terminate (`if n==0 → return TRUE`). So it is not going to depend on N and thus, the running time will be $O(n!)$.

Another case is when the term $O(n!)$ will dominate, i.e., when the number of queens is larger than N, this will happen when there won't be any solution. In this case, the algorithm will take $O(n!)$ time.

In our case, the number of queens is also equal to N. In this case, we can write $O(N^2)\left(O((n-2)!)\right)$ as $\left(O((n-2)!*n*n)\right)$ which is just $\frac{n}{n-1}$ times larger than $n!$ as $\left(\frac{(n-2)!*n*n}{n*(n-1)*(n-2)!} = \frac{n}{n-1}\right)$. According to the definition of Big-Oh, we can choose the value of constant $c > \frac{n}{n-1}$

$(f(n) = O(g(n)),\ if\ f(n) \leq c.\,g(n))$

and thus, say $O(N^2)\ (O((n-2)!)) = O(n!)$. You can use our discussion forum (https://www.codesdope.com/discussion/) to get your doubt cleared.

So by analyzing the equation, we can say that the algorithm is going to take $O(n!)$.

Take a note that this is an optimized version of backtracking algorithm to implement N-Queens (no doubts, it can be further improved). Backtracking - Explanation and N queens problem article (https://www.codesdope.com/blog/article/backtracking-explanation-and-n-queens-problem/) has the non-optimized version of the algorithm, you can compare the running time of the both.

“ Man needs his difficulties because they are necessary to enjoy success. ”

- A. P. J. Abdul Kalam

PREV (/course/algorithms-coin-change/) (/course/algorithms-knights-tour-problem/) NEXT

# Further Readings

- Backtracking - Explanation and N queens problem (/blog/article/backtracking-explanation-and-n-queens-problem/)
- Solving Sudoku with Backtracking | C, Java and Python (/blog/article/solving-sudoku-with-backtracking-c-java-and-python/)
- Backtracking to solve a rat in a maze | C Java Python (/blog/article/backtracking-to-solve-a-rat-in-a-maze-c-java-pytho/)

New Questions

setting up an ide for mac.. - Cpp

(/discussion/setting-up-an-ide-for-mac)