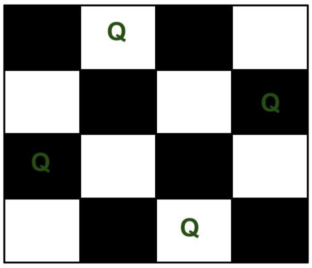
# N Queen Problem | Backtracking-3

We have discussed Knight's tour and Rat in a Maze problems in Set 1 and Set 2 respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

```
{ 0,  1,  0,  0}
{ 0,  0,  0,  1}
{ 1,  0,  0,  0}
{ 0,  0,  1,  0}
```

**Recommended: Please solve it on "*PRACTICE* " first, before moving on to the solution.**

**Naive Algorithm**

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

```
while there are untried configurations
{
   generate the next configuration
   if queens don't attack in this configuration then
   {
      print this configuration;
   }
}
```

**Backtracking Algorithm**

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

```
1) Start in the leftmost column
2) If all queens are placed
    return true
```

```
3) Try all rows in the current column.
   Do following for every tried row.
    a) If the queen can be placed safely in this row
       then mark this [row, column] as part of the
       solution and recursively check if placing
       queen here leads to a solution.
    b) If placing the queen in [row, column] leads to
       a solution then return true.
    c) If placing queen doesn't lead to a solution then
       unmark this [row, column] (Backtrack) and go to
       step (a) to try other rows.
3) If all rows have been tried and nothing worked,
   return false to trigger backtracking.
```

**Implementation of Backtracking solution**

## C/C++

```c
/* C/C++ program to solve N Queen Problem using
   backtracking */
#define N 4
#include <stdbool.h>
#include <stdio.h>

/* A utility function to print solution */
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}

/* A utility function to check if a queen can
   be placed on board[row][col]. Note that this
   function is called when "col" queens are
   already placed in columns from 0 to col -1.
   So we need to check only left side for
   attacking queens */
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    /* Check this row on left side */
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    /* Check upper diagonal on left side */
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    /* Check lower diagonal on left side */
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

/* A recursive utility function to solve N
   Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
    /* base case: If all queens are placed
      then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int i = 0; i < N; i++) {
        /* Check if the queen can be placed on
          board[i][col] */
        if (isSafe(board, i, col)) {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1))
                return true;

            /* If placing queen in board[i][col]
               doesn't lead to a solution, then
               remove queen from board[i][col] */
            board[i][col] = 0; // BACKTRACK
        }
    }

    /* If the queen cannot be placed in any row in
        this colum col  then return false */
    return false;
}

/* This function solves the N Queen problem using
   Backtracking. It mainly uses solveNQUtil() to
   solve the problem. It returns false if queens
   cannot be placed, otherwise, return true and
   prints placement of queens in the form of 1s.
   Please note that there may be more than one
   solutions, this function prints one  of the
   feasible solutions.*/
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

// driver program to test above function
int main()
{
    solveNQ();
    return 0;
}
```

## Java

```
/* Java program to solve N Queen Problem using
   backtracking */
public class NQueenProblem {
    final int N = 4;

    /* A utility function to print solution */
    void printSolution(int board[][])
    {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                System.out.print(" " + board[i][j]
                                 + " ");
            System.out.println();
        }
    }

    /* A utility function to check if a queen can
       be placed on board[row][col]. Note that this
       function is called when "col" queens are already
       placeed in columns from 0 to col -1. So we need
       to check only left side for attacking queens */
    boolean isSafe(int board[][], int row, int col)
    {
        int i, j;

        /* Check this row on left side */
        for (i = 0; i < col; i++)
            if (board[row][i] == 1)
                return false;

        /* Check upper diagonal on left side */
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
            if (board[i][j] == 1)
                return false;

        /* Check lower diagonal on left side */
        for (i = row, j = col; j >= 0 && i < N; i++, j--)
            if (board[i][j] == 1)
                return false;

        return true;
    }

    /* A recursive utility function to solve N
       Queen problem */
    boolean solveNQUtil(int board[][], int col)
    {
        /* base case: If all queens are placed
           then return true */
        if (col >= N)
            return true;

        /* Consider this column and try placing
           this queen in all rows one by one */
        for (int i = 0; i < N; i++) {
            /* Check if the queen can be placed on
               board[i][col] */
            if (isSafe(board, i, col)) {
                /* Place this queen in board[i][col] */
                board[i][col] = 1;

                /* recur to place rest of the queens */
                if (solveNQUtil(board, col + 1) == true)
                    return true;

                /* If placing queen in board[i][col]
                   doesn't lead to a solution then
                   remove queen from board[i][col] */
                board[i][col] = 0; // BACKTRACK
            }
        }

        /* If the queen can not be placed in any row in
           this colum col, then return false */
        return false;
    }

    /* This function solves the N Queen problem using
       Backtracking.  It mainly uses solveNQUtil () to
       solve the problem. It returns false if queens
       cannot be placed, otherwise, return true and
       prints placement of queens in the form of 1s.
       Please note that there may be more than one
       solutions, this function prints one of the
       feasible solutions.*/
    boolean solveNQ()
    {
        int board[][] = { { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 },
                          { 0, 0, 0, 0 } };

        if (solveNQUtil(board, 0) == false) {
            System.out.print("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }

    // driver program to test above function
    public static void main(String args[])
    {
        NQueenProblem Queen = new NQueenProblem();
        Queen.solveNQ();
    }
}
// This code is contributed by Abhishek Shankhadhar
```

### Python3

```
# Python3 program to solve N Queen
# Problem using backtracking
global N
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print (board[i][j], end = " ")
        print()

# A utility function to check if a queen can
# be placed on board[row][col]. Note that this
# function is called when "col" queens are
# already placed in columns from 0 to col -1.
# So we need to check only left side for
# attacking queens
def isSafe(board, row, col):

    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col):

    # base case: If all queens are placed
    # then return true
    if col >= N:
        return True

    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):

        if isSafe(board, i, col):

            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solveNQUtil(board, col + 1) == True:
                return True

            # If placing queen in board[i][col
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0

    # if the queen can not be placed in any row in
    # this colum col then return false
    return False

# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
    board = [ [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0] ]

    if solveNQUtil(board, 0) == False:
        print ("Solution does not exist")
        return False

    printSolution(board)
    return True

# Driver Code
solveNQ()

# This code is contributed by Divyanshu Mehta
```

**Output:** The 1 values indicate placements of queens

```
 0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0
```

**Optimization in is_safe() function**

The idea is not to check every element in right and left diagonal instead use property of diagonals:

1.The sum of i and j is constant and unique for each right diagonal where i is the row of element and j is the column of element.

2.The difference of i and j is constant and unique for each left diagonal where i and j are row and column of element respectively.

**Implementation of Backtracking solution(with optimization)**

## C/C++

```
/* C/C++ program to solve N Queen Problem using
   backtracking */
#define N 4
#include <stdbool.h>
#include <stdio.h>
/* ld is an array where its indices indicate row-col+N-1
 (N-1) is for shifting the difference to store negative
 indices */
int ld[30] = { 0 };
/* rd is an array where its indices indicate row+col
   and used to check whether a queen can be placed on
   right diagonal or not*/
int rd[30] = { 0 };
/*column array where its indices indicates column and
  used to check whether a queen can be placed in that
    row or not*/
int cl[30] = { 0 };
/* A utility function to print solution */
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}

/* A recursive utility function to solve N
   Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
    /* base case: If all queens are placed
      then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int i = 0; i < N; i++) {
        /* Check if the queen can be placed on
          board[i][col] */
        /* A check if a queen can be placed on
           board[row][col].We just need to check
           ld[row-col+n-1] and rd[row+coln] where
           ld and rd are for left and right
           diagonal respectively*/
        if ((ld[i - col + N - 1] != 1 &&
                  rd[i + col] != 1) && cl[i] != 1) {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;
            ld[i - col + N - 1] =
                          rd[i + col] = cl[i] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1))
                return true;

            /* If placing queen in board[i][col]
               doesn't lead to a solution, then
               remove queen from board[i][col] */
            board[i][col] = 0; // BACKTRACK
            ld[i - col + N - 1] =
                         rd[i + col] = cl[i] = 0;
        }
    }

    /* If the queen cannot be placed in any row in
        this colum col  then return false */
    return false;
}
/* This function solves the N Queen problem using
   Backtracking. It mainly uses solveNQUtil() to
   solve the problem. It returns false if queens
   cannot be placed, otherwise, return true and
   prints placement of queens in the form of 1s.
   Please note that there may be more than one
   solutions, this function prints one  of the
   feasible solutions.*/
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

// driver program to test above function
int main()
{
    solveNQ();
    return 0;
}
```

### Python3

```
""" Python3 program to solve N Queen Problem using
backtracking """
N = 4

""" ld is an array where its indices indicate row-col+N-1
(N-1) is for shifting the difference to store negative
indices """
ld = [0] * 30

""" rd is an array where its indices indicate row+col
and used to check whether a queen can be placed on
right diagonal or not"""
rd = [0] * 30

"""column array where its indices indicates column and
used to check whether a queen can be placed in that
    row or not"""
cl = [0] * 30

""" A utility function to print solution """
def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end = " ")
        print()

""" A recursive utility function to solve N
Queen problem """
def solveNQUtil(board, col):

    """ base case: If all queens are placed
        then return True """
    if (col >= N):
        return True

    """ Consider this column and try placing
        this queen in all rows one by one """
    for i in range(N):

        """ Check if the queen can be placed on board[i][col] """
        """ A check if a queen can be placed on board[row][col].
        We just need to check ld[row-col+n-1] and rd[row+coln]
        where ld and rd are for left and right diagonal respectively"""
        if ((ld[i - col + N - 1] != 1 and
             rd[i + col] != 1) and cl[i] != 1):
```

```
            """ Place this queen in board[i][col] """
            board[i][col] = 1
            ld[i - col + N - 1] = rd[i + col] = cl[i] = 1

            """ recur to place rest of the queens """
            if (solveNQUtil(board, col + 1)):
                return True

            """ If placing queen in board[i][col]
            doesn't lead to a solution,
            then remove queen from board[i][col] """
            board[i][col] = 0 # BACKTRACK
            ld[i - col + N - 1] = rd[i + col] = cl[i] = 0

            """ If the queen cannot be placed in
            any row in this colum col then return False """
    return False

""" This function solves the N Queen problem using
Backtracking. It mainly uses solveNQUtil() to
solve the problem. It returns False if queens
cannot be placed, otherwise, return True and
prints placement of queens in the form of 1s.
Please note that there may be more than one
solutions, this function prints one of the
feasible solutions."""
def solveNQ():
    board = [[0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0]]
    if (solveNQUtil(board, 0) == False):
        printf("Solution does not exist")
        return False
    printSolution(board)
    return True

# Driver Code
solveNQ()

# This code is contributed by SHUBHAMSINGH10
```

**Output:** The 1 values indicate placements of queens

```
 0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0
```

Printing all solutions in N-Queen Problem

**Sources:**

http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf

http://en.literateprograms.org/Eight_queens_puzzle_%28C%29

http://en.wikipedia.org/wiki/Eight_queens_puzzle

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Recommended Posts:**

8 queen problem

Printing all solutions in N-Queen Problem

N Queen Problem using Branch And Bound

N Queen in O(n) space

Check if a Queen can attack a given cell on chessboard

Number of cells a queen can move with obstacles on the chessborad

Nuts & Bolts Problem (Lock & Key problem) | Set 2 (Hashmap)

Nuts & Bolts Problem (Lock & Key problem) | Set 1

Job Sequencing Problem

Tiling Problem

Partition problem | DP-18

The Celebrity Problem

Box Stacking Problem | DP-22

Subset Sum Problem | DP-25

0-1 Knapsack Problem | DP-10

**Improved By :** AniruddhaPandey, Parimal7, harminder3027, SHUBHAMSINGH10

**Article Tags :** Backtracking Accolite Amazon Amdocs chessboard-problems MAQ Software Twitter Visa

**Practice Tags :** Amazon Accolite Visa MAQ Software Amdocs Twitter Backtracking

23

3.5 ▲

□ To-do □ Done

Based on **214** vote(s)

Feedback/ Suggest Improvement Add Notes Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

▲