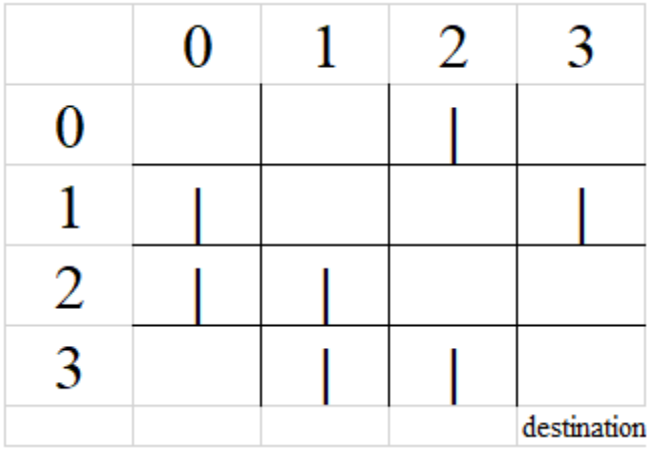
BACKTRACKING

What is Back Tracking?

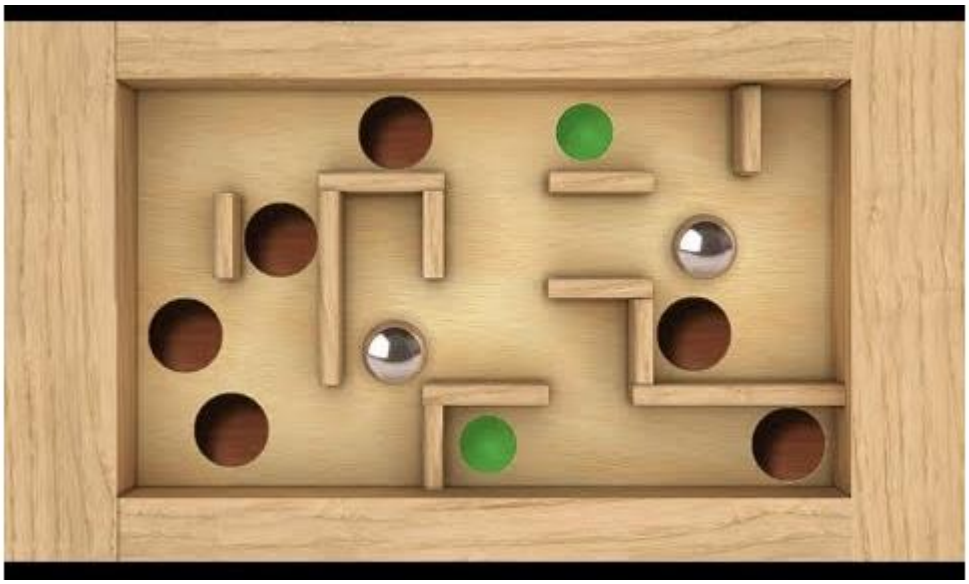
Backtracking *is an algorithmic-technique for* ***solving problems recursively*** *by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).*

Suppose we have a gird,



Now, you want to move from (0, 0) to (3, 3), You need to find your way through this grid by avoiding the blockage / walls and reach your destination.

Just like a maze game,



In the first step, we move from (0,0) to (0,1). Then we move from (0,1) to (0,2). At (0,2) we have a wall, so we will back track from (0,2) to (0,1).

Now, from (0,1) to (1,1) and from (1,1) to (2,1). Since we have a wall at (2,1), we will back track again to (1,1). These steps would be repeated until we reach the destination. We will explore all paths and if there is a blockage, we will back track to the previous position, this is the logic behind backtracking. It is recursion only, and not a greedy algorithm.

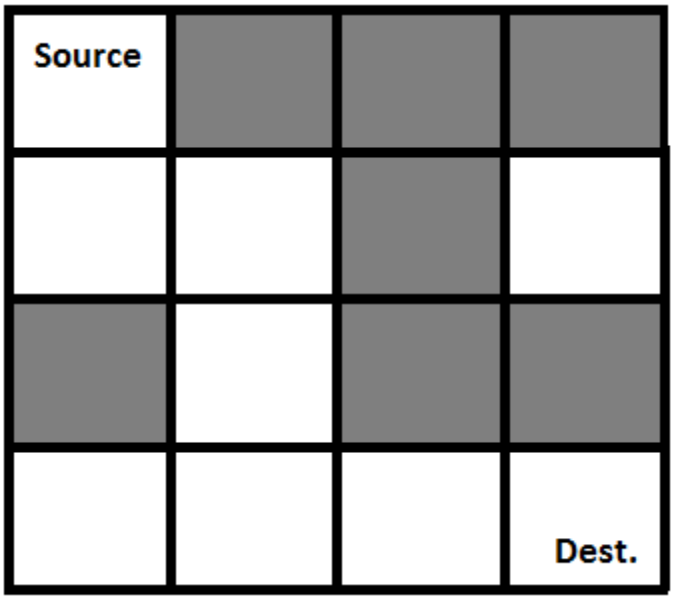
IN backtracking is a type of algorithm, where we try to explore all states and if that state is not giving us valid solution then we come back.

Rat in a Maze | Backtracking – 2

A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Following is an example maze.
Gray blocks are dead ends (value = 0).



Following is binary matrix representation of the above maze.

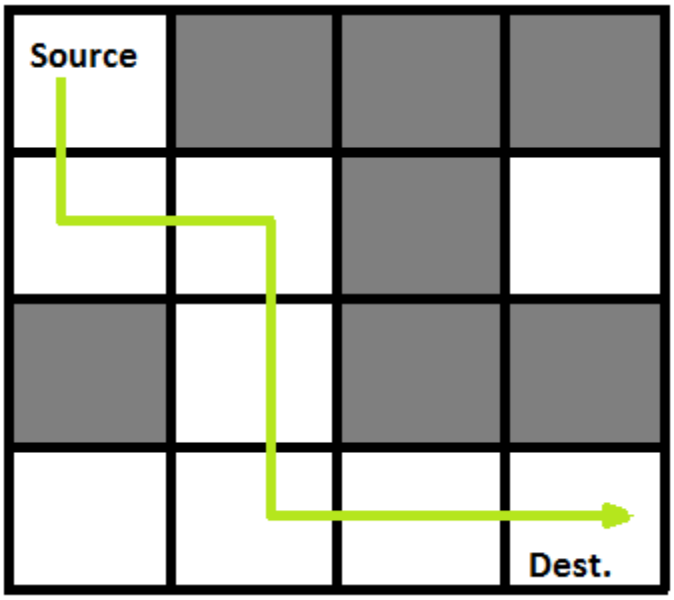
{1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

Following is a maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrix.

{1, 0, 0, 0}

{1, 1, 0, 0}

{0, 1, 0, 0}

{0, 1, 1, 1}

All entries in solution path are marked as 1.

Backtracking Algorithm: Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally. Solving one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree) is the process of backtracking.

Approach: Form a recursive function, which will follow a path and check if the path reaches the destination or not. If the path does not reach the destination then backtrack and try other paths.

Algorithm:

1. Create a solution matrix, initially filled with 0's.
2. Create a recursive function, which takes initial matrix, output matrix and position of rat (i, j).
3. if the position is out of the matrix or the position is not valid then return.
4. Mark the position output[i][j] as 1 and check if the current position is destination or not. If destination is reached print the output matrix and return.
5. Recursively call for position (i+1, j) and (i, j+1).
6. Unmark position (i, j), i.e output[i][j] = 0.

CODE:

```
# Python3 program to solve Rat in a Maze
# problem using backracking

# Maze size
N = 4

# A utility function to print solution matrix sol
def printSolution( sol ):

    for i in sol:
        for j in i:
            print(str(j) + " ", end ="")
        print("")

# A utility function to check if x, y is valid
# index for N * N Maze
def isSafe( maze, x, y ):

    if x >= 0 and x < N and y >= 0 and y < N and maze[x][y] == 1:
        return True

    return False

""" This function solves the Maze problem using Backtracking. It mainly
    uses solveMazeUtil() to solve the problem. It returns false if no
    path is possible, otherwise return true and prints the path in the
    form of 1s. Please note that there may be more than one solutions,
    this function prints one of the feasable solutions. """

def solveMaze( maze ):

    # Creating a 4 * 4 2-D list
    sol = [ [ 0 for j in range(4) ] for i in range(4) ]

    if solveMazeUtil(maze, 0, 0, sol) == False:
        print("Solution doesn't exist");
        return False

    printSolution(sol)
    return True

# A recursive utility function to solve Maze problem
def solveMazeUtil(maze, x, y, sol):
```

```
    # if (x, y is goal) return True
    if x == N - 1 and y == N - 1 and maze[x][y]== 1:
        sol[x][y] = 1
        return True

    # Check if maze[x][y] is valid
    if isSafe(maze, x, y) == True:
        # mark x, y as part of solution path
        sol[x][y] = 1

        # Move forward in x direction
        if solveMazeUtil(maze, x + 1, y, sol) == True:
            return True

        # If moving in x direction doesn't give solution
        # then Move down in y direction
        if solveMazeUtil(maze, x, y + 1, sol) == True:
            return True

        # If none of the above movements work then
        # BACKTRACK: unmark x, y as part of solution path
        sol[x][y] = 0
        return False

# Driver program to test above function
if __name__ == "__main__":
    # Initialising the maze
    maze = [ [1, 0, 0, 0],
             [1, 1, 0, 1],
             [0, 1, 0, 0],
             [1, 1, 1, 1] ]

    solveMaze(maze)
```

Backtracking Problems:

- The Knight's tour problem
- Rat in a Maze
- Subset Sum
- m Colouring Problem)
- Sudoku

LeetCode: 39. Combination Sum

The given array's are in sorted arrays and furthermore, the outputs are also sorted.

Input: candidates = [2, 3, 5], target = 8

Output: [[2, 2, 2, 2], [2, 3, 3], [3, 5]

The frequency of each output is,

[2, 2, 2, 2] → 2**:** 4

[2, 3, 3] → 2:1, 3:2

[3, 5] → 3:1, 5:1

Implementation:

```
class Solution:
    def generateSolution(self, idx, temp):
        if sum(temp) > self.target:
            return

        if sum(temp) == self.target:
            ans = sorted(temp[:])
            ans = ",".join(map(str, ans))
            self.res.add(ans)
            return

        for i in range(idx, len(self.candidates)):
            temp.append(self.candidates[i])
            self.generateSolution(idx, temp)
            temp.pop() # backtracking

    def combinationSum(self, candidates: List[int], target: int) -
> List[List[int]]:

        self.candidates = candidates
        self.target = target

        self.res = set()

        # The first argument is the index and second argument is the tem
porary list.
        self.generateSolution(0, [])

        ans = list()
```

```
        for s in self.res:
            ans.append(map(int, s.split(",")))

        return ans
```

Time Complexity of Backtracking will always be exponent (k^n). Hence Backtracking is very worst.

LeetCode: 51. N Queens

```
class Solution:
    def safe(self, row, col, chessboard):
        # Diagnoly right up
        r = row
        c = col
        while r >= 0 and c < self.n:
            if chessboard[r][c] == 'Q':
                return False
            r -= 1
            c += 1

        # Diagnoly right down
        r = row
        c = col
        while r < self.n and c < self.n:
            if chessboard[r][c] == 'Q':
                return False
            r += 1
            c += 1

        # Diagnoly left up
        r = row
        c = col
        while r >= 0 and c >= 0:
            if chessboard[r][c] == 'Q':
                return False
            r -= 1
            c -= 1

        # Diagnoly left down
        r = row
```

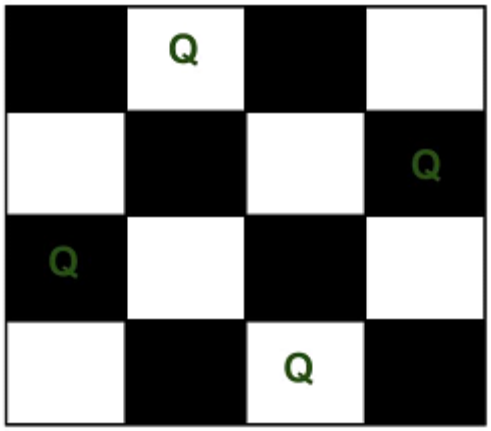
```
        c = col
        while r >= self.n and c >= 0:
            if chessboard[r][c] == 'Q':
                return False
            r += 1
            c -= 1

        # horizontally right
        r = row
        c = col
        while c < self.n:
            if chessboard[r][c] == 'Q':
                return False
            c += 1

        # horizontally left
        r = row
        c = col
        while c >= 0:
            if chessboard[r][c] == 'Q':
                return False
            c -= 1

        # vertically up
        r = row
        c = col
        while r >= 0:
            if chessboard[r][c] == 'Q':
                return False
            r -= 1

        # vertically down
        r = row
        c = col
        while r < self.n:
            if chessboard[r][c] == 'Q':
                return False
            r += 1
        return True

    def solve(self, row, chessboard):
        if row == self.n:
            temp = list()
            for row in chessboard:
```

```
                temp.append("".join(row))
            self.res.append(temp)
            return

        for col in range(0, self.n):
            if self.safe(row, col, chessboard):
                chessboard[row][col] = 'Q'
                self.solve(row + 1, chessboard)
                chessboard[row][col] = '.'

    def solveNQueens(self, n: int) -> List[List[str]]:
        self.n = n
        self.res = list()

        chessboard = [["." for _ in range(self.n)] for _ in range(self.n
)]
        self.solve(0, chessboard)

        return self.res
```

Other Approaches from GeekforGeeks: N Queen Problem

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}

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

```
# 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 Backtraking. It mainly
# uses solveNQUtil() to solve the problem. It return 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 solution, 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()
```

Output: The 1 value 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)

```
""" 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 indicate 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()
```

Output: The 1 values indicate placements of queens

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

Problems on N-Queen:

https://www.geeksforgeeks.org/printing-solutions-n-queen-problem/?ref=rp

https://www.geeksforgeeks.org/n-queen-problem-local-search-using-hill-climbing-with-random-neighbour/?ref=rp

https://www.geeksforgeeks.org/n-queen-problem-using-branch-and-bound/?ref=rp

https://www.geeksforgeeks.org/n-queen-in-on-space/?ref=rp