Backtracking

Introduction

Backtracking is basically an approach to explore the paths that a problem can follow. It is implemented via recursion.
Thus, we can define backtracking as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

For example, we have 3 bags. Your friend has put a ball inside one of these bags, and has tasked you with finding the bag that contains the ball.

The idea of backtracking suggests that we explore all the possibilities to find the ball. We first look for the ball in bag 1. If the ball is found, all is well and good. If not, we now look for the ball in bag 2. We continue this process till the point where we either find the ball, or we have checked all the bags.

Why Backtracking?

For every problem with clear and well-defined constraints on any solution, Backtracking incrementally builds a candidate to the solution and abandons the candidate (" begins backtrack") as soon as it determines that the candidate can not be completed to a correct solution. Most problems can be solved using other algorithms like Greedy Algorithms, Dynamic Programming in O(Nlog(N), O(N) and O(log(N)) time complexity, and hence outshine the backtracking algorithm (generally, backtracking algorithms have exponential

time and space complexity). However, for a few problems, we only have the backtracking algorithms to solve until now.

Let's discuss a problem to understand the benefit of backtracking approach:

The Knight's Tour

Problem Statement:
Given a N*N board with the Knight placed on the first block of an empty board. Moving according to the rules of chess, knights must visit each square exactly once. Print the order of each cell in which they are visited.

Example:
Input :
N = 8
Output:
0 59 38 33 30 17 8 63
37 34 31 60 9 62 29 16
58 1 36 39 32 27 18 7
35 48 41 26 61 10 15 28
42 57 2 49 40 23 6 19
47 50 45 54 25 20 11 14
56 43 52 3 22 13 24 5
51 46 55 44 53 4 21 12

Naive Approach: Generate all paths one by one and check if the generated path satisfies the constraints. That is, while there are untried paths, generate

the next path and if this path covers every square on the chess board then print this path.

Backtracking Approach: Backtracking incrementally attacks a problem.

- We start with an empty solution vector and add Knight's move(item) one by one.
- We check whether the current move violates the problem constraint, by using a recursion call.
- If recursion tells us that the solution is not valid, then we remove the previous move(item) and try an alternative move(item).
- If none of the alternative moves(items) work then we go to the previous stage and remove the Knight's move(item) added in the previous stage.
- If we reach the initial stage, that means there is no solution, However if the solution vector becomes complete then we print the solution.

N- Queen Problem

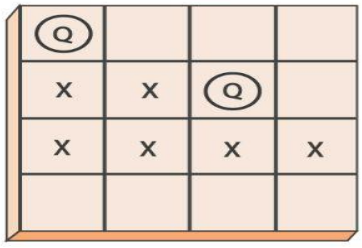
Problem Statement: We are given an **N X N Matrix** and we have to place N Queens in it such that no Queens cross each other .
A Queen can move in all eight directions.

Explanation:
Applying Backtracking:

1. **Place yourself at the start point.**
2. **Explore all the paths possible from that starting point.**
3. **Insert your element at only correct positions and once inserted move on to a smaller problem.**
4. **Once you reach the base case, print your solution.**
5. **Once you explore all the paths from one point , backtrack to your previous point.**

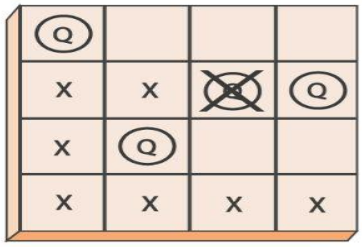
CODING NINJAS

N-Queen Problem:-

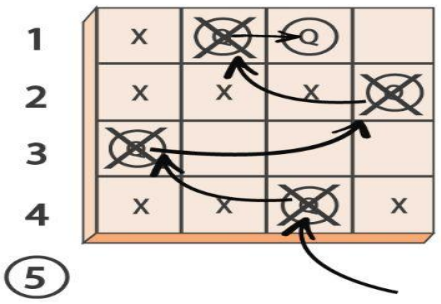


Possible Position

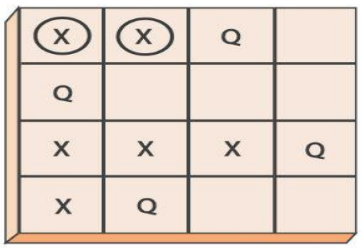
No, Queen can be inserted in 3rd row,
So Backtrack



Again Backtrack x 2



row > N ∘ i can print
this solution



This is also a solution
then again backtrack

Code:

```
#include <bits/stdc++.h>

using namespace std;

int board[11][11];

bool isPossible(int n,int row,int col){

// Same Column
  for(int i=row-1;i>=0;i--){
    if(board[i][col] == 1){
      return false;
    }
  }
//Upper Left Diagonal
  for(int i=row-1,j=col-1;i>=0 && j>=0 ; i--,j--){
    if(board[i][j] ==1){
      return false;
    }
  }

  // Upper Right Diagonal

  for(int i=row-1,j=col+1;i>=0 && j<n ; i--,j++){
    if(board[i][j] == 1){
      return false;
    }
  }

  return true;
}
void nQueenHelper(int n,int row){
  if(row==n){
    // We have reached some solution.
    // Print the board matrix
    // return

    for(int i=0;i<n;i++){
```

```
      for(int j=0;j<n;j++){
        cout << board[i][j] << " ";
      }
    }
    cout<<endl;
    return;

  }

  // Place at all possible positions and move to smaller problem
  for(int j=0;j<n;j++){

    if(isPossible(n,row,j)){
      board[row][j] = 1;
      nQueenHelper(n,row+1);
      board[row][j] = 0;
    }
  }
  return;

}
void placeNQueens(int n){

  memset(board,0,11*11*sizeof(int));

  nQueenHelper(n,0);

}
int main(){

  placeNQueens(4);
  return 0;
}
#include <bits/stdc++.h>

using namespace std;

int board[11][11];

bool isPossible(int n,int row,int col){
```

```
// Same Column
  for(int i=row-1;i>=0;i--){
    if(board[i][col] == 1){
      return false;
    }
  }
//Upper Left Diagonal
  for(int i=row-1,j=col-1;i>=0 && j>=0 ; i--,j--){
    if(board[i][j] ==1){
      return false;
    }
  }

  // Upper Right Diagonal

  for(int i=row-1,j=col+1;i>=0 && j<n ; i--,j++){
    if(board[i][j] == 1){
      return false;
    }
  }

  return true;
}
void nQueenHelper(int n,int row){
  if(row==n){
    // We have reached some solution.
    // Print the board matrix
    // return

    for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
        cout << board[i][j] << " ";
      }
    }
    cout<<endl;
    return;

  }
```

```
  // Place at all possible positions and move to smaller problem
  for(int j=0;j<n;j++){

    if(isPossible(n,row,j)){
      board[row][j] = 1;
      nQueenHelper(n,row+1);
      board[row][j] = 0;
    }
  }
  return;

}
void placeNQueens(int n){

  memset(board,0,11*11*sizeof(int));

  nQueenHelper(n,0);

}
int main(){

  placeNQueens(4);
  return 0;
}
```

Rat In A Maze Problem

Problem Statement: We are given an **N X N Matrix** and we have a starting point and a destination point. There is a rat who has to reach the destination point from the starting point through a possible path and we have to print this path.

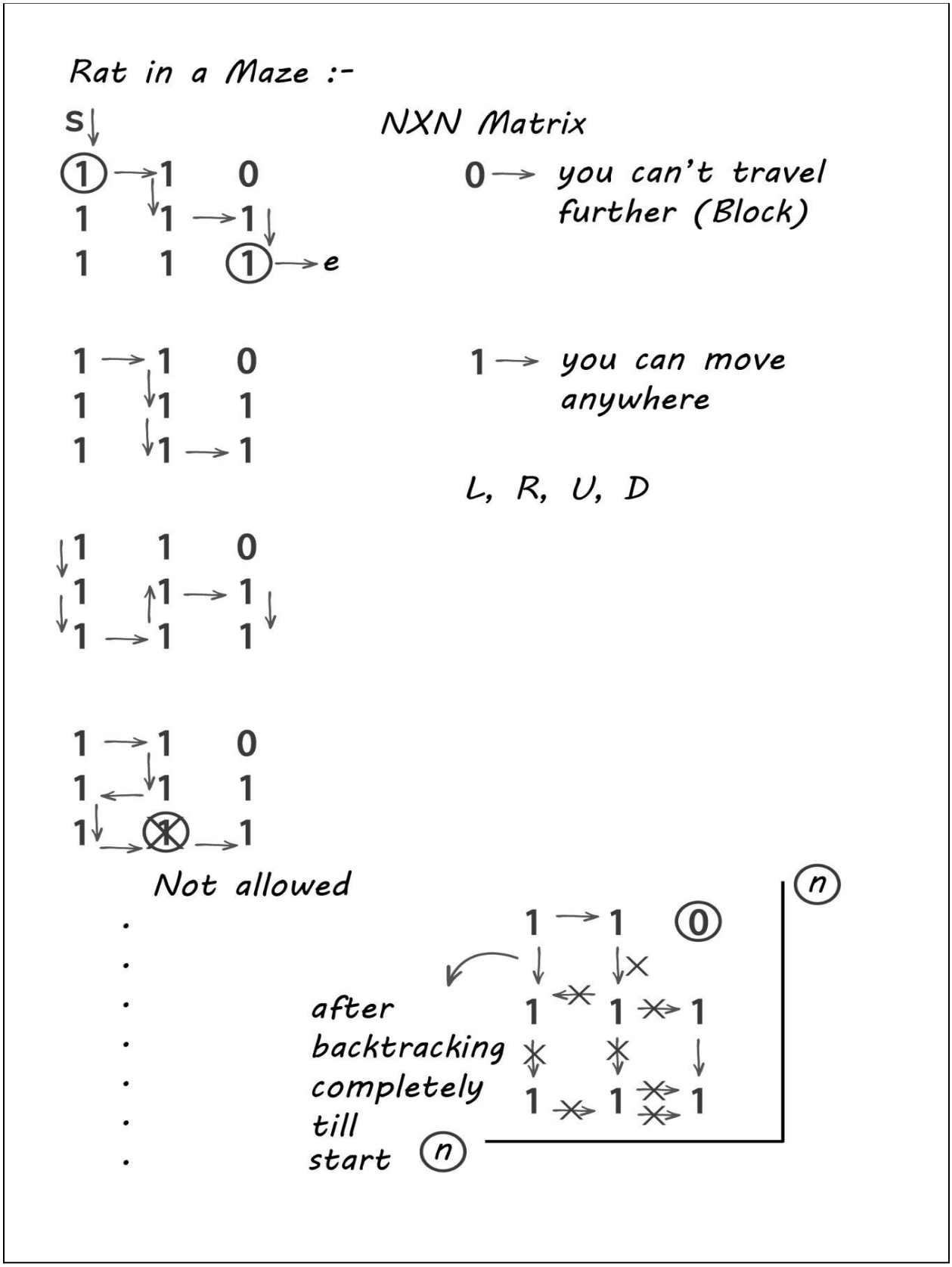
Matrix =

1	**1**	**0**
1	**1**	**1**
1	**1**	**1**

0 means that the rat can't travel further.
1 means we can move anywhere from that point, that is, left, right, up and down.

Explanation:

1. **Create a solution matrix, initially filled with 0's.**
2. **Create a recursive function, which takes the 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 the destination is reached, print the output matrix and return.**
5. **Call recursion at position (i+1, j) and (i, j+1).**
6. **Unmark position (i, j), i.e output[i][j] = 0.**


Rat in a Maze :-
S
NXN Matrix
0 → you can't travel further (Block)
1 → you can move anywhere
L, R, U, D
e
Not allowed
after backtracking completely till start
n
0

Code:

```
#include<bits/stdc++.h>
using namespace std;


void printSolution(int** solution,int n){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            cout << solution[i][j] << " ";
        }
    }
    cout<<endl;
}
void mazeHelp(int maze[][20],int n,int** solution,int x,int y){


    if(x == n-1 && y == n-1){
        solution[x][y] =1;
        printSolution(solution,n);
        solution[x][y] = 0;
        return;
    }
    if(x>=n || x<0 || y>=n || y<0 || maze[x][y] ==0 || solution[x][y] ==1){
        return;
    }
    solution[x][y] = 1;
    mazeHelp(maze,n,solution,x-1,y);
    mazeHelp(maze,n,solution,x+1,y);
    mazeHelp(maze,n,solution,x,y-1);
    mazeHelp(maze,n,solution,x,y+1);
    solution[x][y] = 0;
}
void ratInAMaze(int maze[][20], int n){
  int** solution = new int*[n];
  for(int i=0;i<n;i++){
    solution[i] = new int[n];
  }
  mazeHelp(maze,n,solution,0,0); }
```