Print All Permutations of a String/Array

Problem Statement: Given an array arr of distinct integers, print all permutations of String/Array.

Examples:

```
Example 1:
Input: arr = [1, 2, 3]
Output:
[
  [1, 2, 3],
  [1, 3, 2],
  [2, 1, 3],
  [2, 3, 1],
  [3, 1, 2],
  [3, 2, 1]
]
Explanation: Given an array, return all the possible permutations.
Example 2:
Input: arr = [0, 1]
Output:
[
  [0, 1],
  [1, 0]
]
Explanation: Given an array, return all the possible permutations.
```

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Recursive

Approach: We have given the nums array, so we will declare an ans vector of vector that will store all the permutations also declare a data structure.

Declare a map and initialize it to zero and call the recursive function

Base condition:

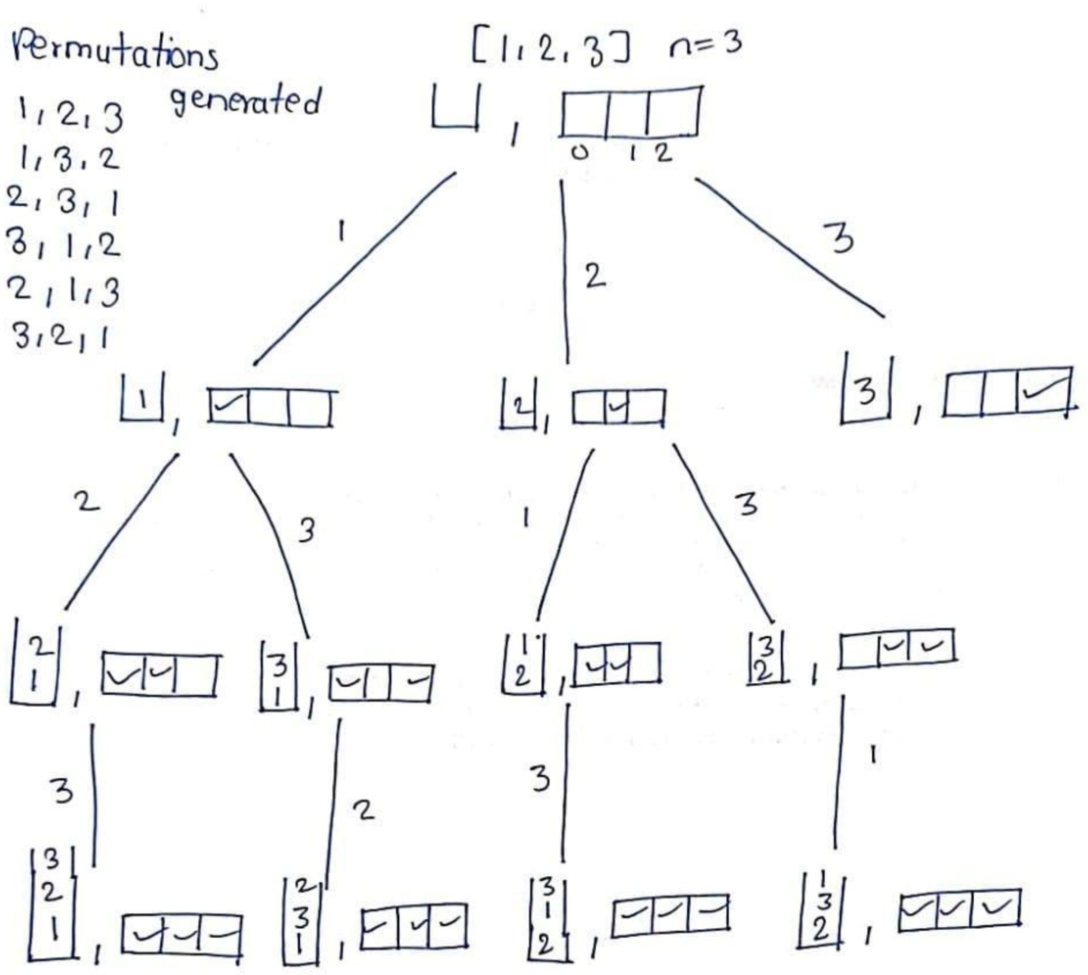
When the data structure's size is equal to n(size of nums array) then it is a permutation and stores that permutation in our ans, then returns it.

Recursion:

Run a for loop starting from 0 to nums.size() – 1. Check if the frequency of i is unmarked, if it is unmarked then it means it has not been picked and then we pick. And make sure it is marked as picked.

Call the recursion with the parameters to pick the other elements when we come back from the recursion make sure you throw that element out. And unmark that element in the map.

Recursive Tree:



Code:
```
#include<bits/stdc++.h>
using namespace std;
class Solution {
  private:
    void recurPermute(vector < int > & ds, vector < int > & nums,
vector < vector < int >> & ans, int freq[]) {
      if (ds.size() == nums.size()) {
        ans.push_back(ds);
        return;
      }
      for (int i = 0; i < nums.size(); i++) {
        if (!freq[i]) {
          ds.push_back(nums[i]);
          freq[i] = 1;
          recurPermute(ds, nums, ans, freq);
          freq[i] = 0;
          ds.pop_back();
        }
      }
    }
  public:
    vector < vector < int >> permute(vector < int > & nums) {
      vector < vector < int >> ans;
      vector < int > ds;
      int freq[nums.size()];
      for (int i = 0; i < nums.size(); i++) freq[i] = 0;
      recurPermute(ds, nums, ans, freq);
      return ans;
    }
};

int main() {
  Solution obj;
  vector<int> v{1,2,3};
  vector < vector < int >> sum = obj.permute(v);
  cout << "All Permutations are " << endl;
  for (int i = 0; i < sum.size(); i++) {
    for (int j = 0; j < sum[i].size(); j++)
      cout << sum[i][j] << " ";
    cout << endl;
  }
}
```

Output:

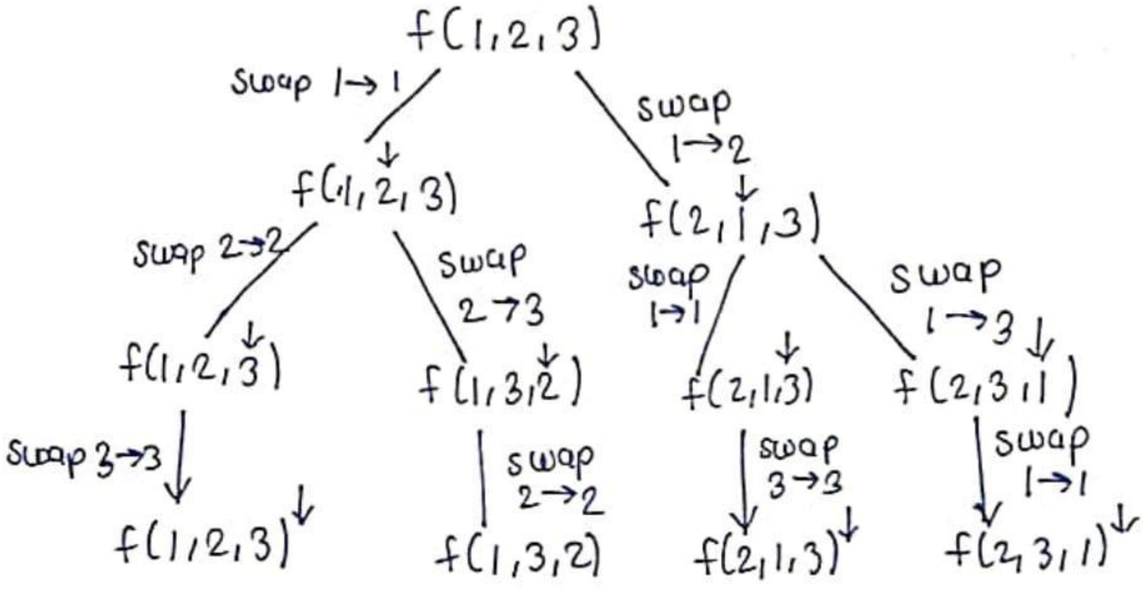
```
All Permutations are
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Time Complexity: N! x N
Space Complexity: O(N)

Solution 2: With Backtracking.
Approach: Using backtracking to solve this.

We have given the nums array, so we will declare an ans vector of vector that will store all the permutations.
Call a recursive function that starts with zero, nums array, and ans vector.
Declare a map and initialize it to zero and call the recursive function

Base condition:
Whenever the index reaches the end take the nums array and put it in ans vector and return.
Recursion:
Go from index to n – 1 and swap. Once the swap has been done call recursion for the next state.After coming back from the recursion make sure you re-swap it because for the next element the swap will not take place.
Recursive Tree:



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

using namespace std;
class Solution {
  private:
    void recurPermute(int index, vector < int > & nums, vector <
vector < int >> & ans) {
      if (index == nums.size()) {
        ans.push_back(nums);
        return;
      }
      for (int i = index; i < nums.size(); i++) {
        swap(nums[index], nums[i]);
        recurPermute(index + 1, nums, ans);
        swap(nums[index], nums[i]);
      }
    }
  public:
    vector < vector < int >> permute(vector < int > & nums) {
      vector < vector < int >> ans;
      recurPermute(0, nums, ans);
      return ans;
    }
};

int main() {
  Solution obj;
  vector < int > v {1,2,3};
  vector < vector < int >> sum = obj.permute(v);
  cout << "All Permutations are" << endl;
  for (int i = 0; i < sum.size(); i++) {
    for (int j = 0; j < sum[i].size(); j++)
      cout << sum[i][j] << " ";
    cout << endl;
  }
}
```

Output:

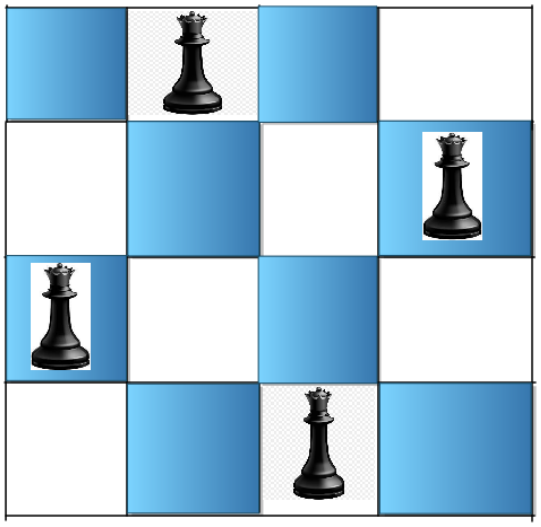
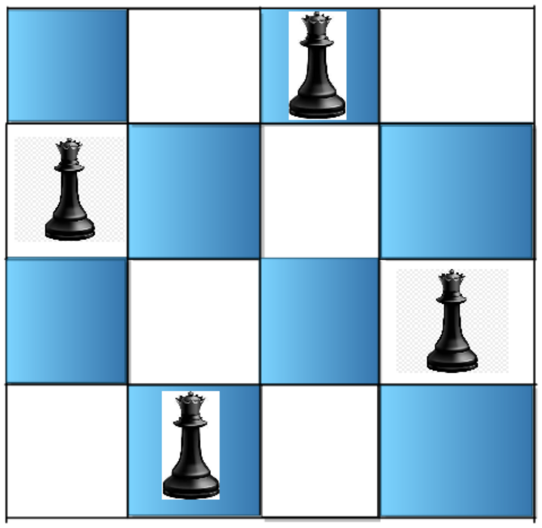
```
All Permutations are
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Time Complexity: O(N! X N)
Space Complexity: O(1)

N Queen Problem | Return all Distinct Solutions to the N-Queens Puzzle

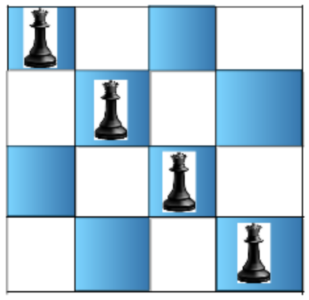
Problem Statement: The n-queens is the problem of placing n queens on n × n chessboard such that no two queens can attack each other. Given an integer n, return all distinct solutions to the n -queens puzzle. Each solution contains a distinct boards configuration of the queen's placement, where 'Q' and '.' indicate queen and empty space respectively.

```
Examples:
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens
puzzle as shown below
```

Two arrangements possible for 4 queens

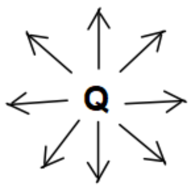
Let us first understand how can we place queens in a chessboard so that no attack on either of them can take place.



Rules for n-Queen in chessboard

1. Every row should have one Queen
2. Every column should have one Queen
3. No two queens can attack each other

Queen attack can take place in following way

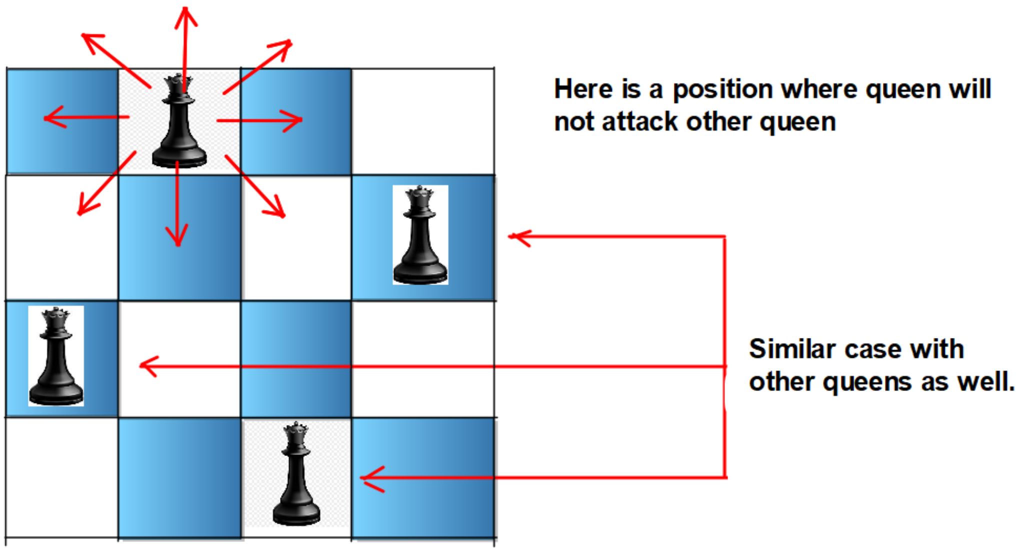


Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

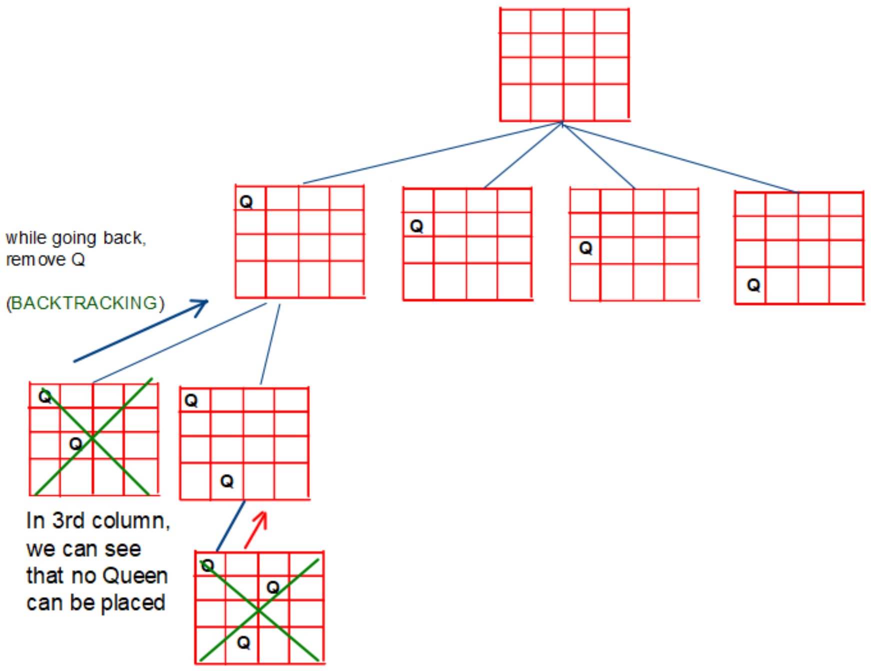
Solution 1:

Intuition: Using the concept of Backtracking, we will place Queen at different positions of the chessboard and find the right arrangement where all the n queens can be placed on the n*n grid.

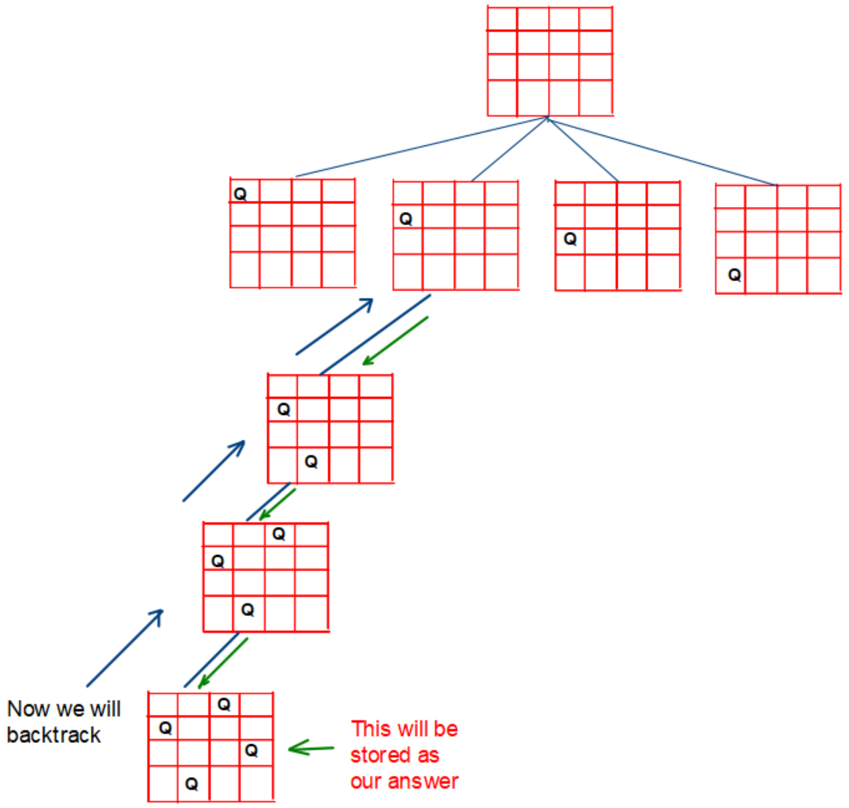

Here is a position where queen will not attack other queen
Similar case with other queens as well.

Approach:

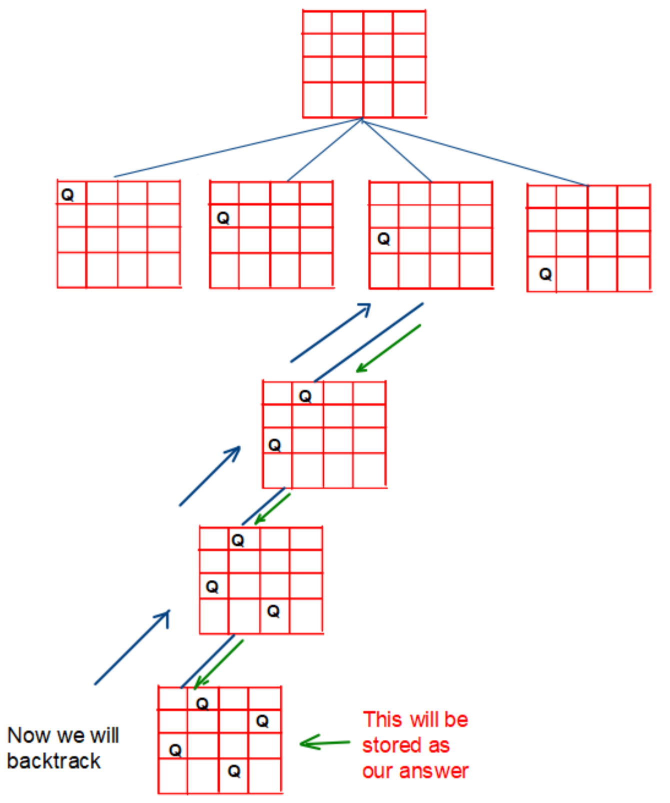
Ist position: This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 3rd column, the Queen will be killed at all possible positions of row.

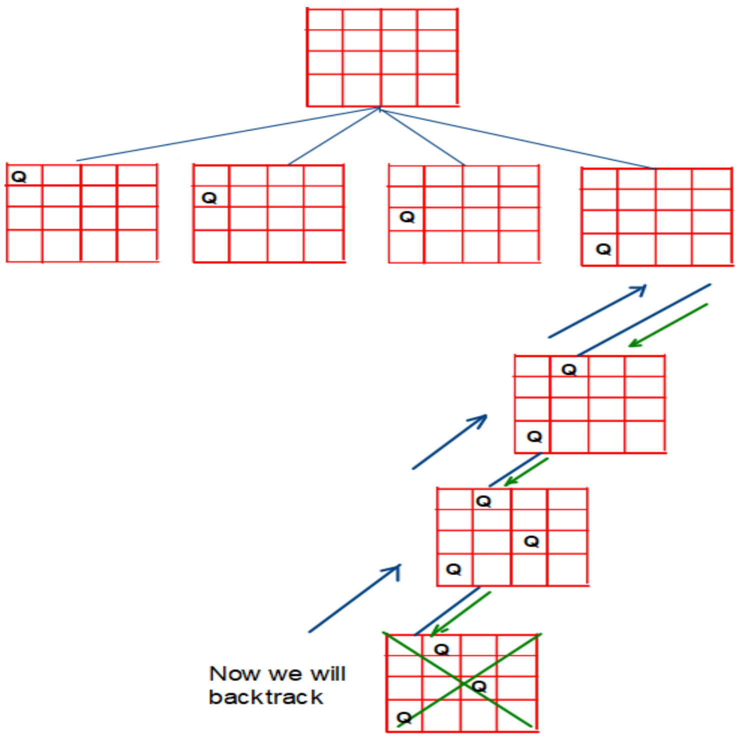


2nd position: One of the correct possible arrangements is found. So we will store it as our answer.



3rd position: One of the correct possible arrangements is found. So we will store it as our answer.

	**4th position:** This is the position where we can see no possible arrangement is found where all queens can be placed since, at the 4th column, the Queen will be killed at all possible positions of row.



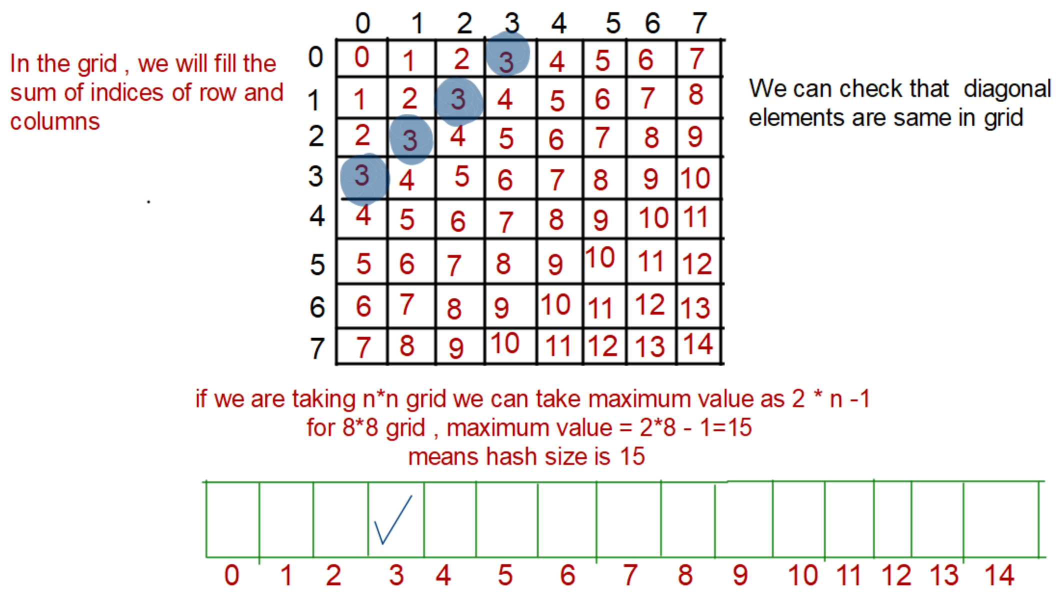
Code:
```
#include <bits/stdc++.h>

using namespace std;
class Solution {
  public:
    bool isSafe1(int row, int col, vector < string > board, int n) {
      // check upper element
      int duprow = row;
      int dupcol = col;

      while (row >= 0 && col >= 0) {
        if (board[row][col] == 'Q')
          return false;
        row--;
        col--;
      }

      col = dupcol;
      row = duprow;
      while (col >= 0) {
        if (board[row][col] == 'Q')
          return false;
        col--;
      }

      row = duprow;
      col = dupcol;
      while (row < n && col >= 0) {
        if (board[row][col] == 'Q')
          return false;
        row++;
        col--;
      }
      return true;
    }

  public:
    void solve(int col, vector < string > & board, vector < vector <
string >> & ans, int n) {
      if (col == n) {
        ans.push_back(board);
        return;
      }
      for (int row = 0; row < n; row++) {
        if (isSafe1(row, col, board, n)) {
          board[row][col] = 'Q';
          solve(col + 1, board, ans, n);
          board[row][col] = '.';
        }
      }
    }

  public:
    vector < vector < string >> solveNQueens(int n) {
      vector < vector < string >> ans;
      vector < string > board(n);
      string s(n, '.');
      for (int i = 0; i < n; i++) {
        board[i] = s;
      }
      solve(0, board, ans, n);
      return ans;
    }
};
int main() {
  int n = 4; // we are taking 4*4 grid and 4 queens
  Solution obj;
  vector < vector < string >> ans = obj.solveNQueens(n);
  for (int i = 0; i < ans.size(); i++) {
    cout << "Arrangement " << i + 1 << "\n";
    for (int j = 0; j < ans[0].size(); j++) {
      cout << ans[i][j];
      cout << endl;
    }
    cout << endl;
  }
  return 0;
}
```

Output:

```
Arrangement 1
..Q.
Q…
…Q
.Q..
Arrangement 2
.Q..
…Q
Q…
..Q.
```

Time Complexity: Exponential in nature, since we are trying out all ways. To be precise it goes as O
(N! * N) nearly.
Space Complexity: O(N^2)
Solution 2:
Intuition: This is the optimization of the issafe function. In the previous issafe function, we need o(N) for a row, o(N) for the column, and o(N) for diagonal. Here, we will use hashing to maintain a list to check whether that position can be the right one or not.
Approach: For checking Left row elements

For checking Left row

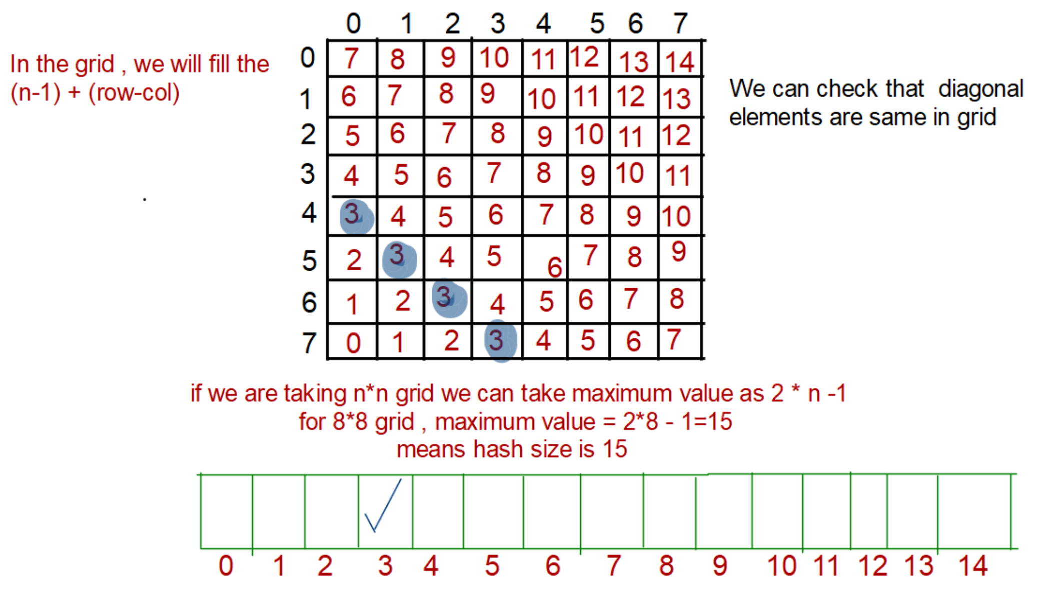

In the grid , we will fill the sum of indices of row and columns
We can check that diagonal elements are same in grid
if we are taking n*n grid we can take maximum value as 2 * n -1
for 8*8 grid , maximum value = 2*8 - 1=15
means hash size is 15

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	9
3	3	4	5	6	7	8	9	10
4	4	5	6	7	8	9	10	11
5	5	6	7	8	9	10	11	12
6	6	7	8	9	10	11	12	13
7	7	8	9	10	11	12	13	14

			✓											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

For checking upper diagonal and lower diagonal

For checking upper diagonal and lower diagonal


In the grid , we will fill the (n-1) + (row-col)
We can check that diagonal elements are same in grid
if we are taking n*n grid we can take maximum value as 2 * n -1
for 8*8 grid , maximum value = 2*8 - 1=15
means hash size is 15

	0	1	2	3	4	5	6	7
0	7	8	9	10	11	12	13	14
1	6	7	8	9	10	11	12	13
2	5	6	7	8	9	10	11	12
3	4	5	6	7	8	9	10	11
4	3	4	5	6	7	8	9	10
5	2	3	4	5	6	7	8	9
6	1	2	3	4	5	6	7	8
7	0	1	2	3	4	5	6	7

			✓											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

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

using namespace std;
class Solution {
  public:
    void solve(int col, vector < string > & board, vector < vector <
string >> & ans, vector < int > & leftrow, vector < int > &
upperDiagonal, vector < int > & lowerDiagonal, int n) {
      if (col == n) {
        ans.push_back(board);
        return;
      }
      for (int row = 0; row < n; row++) {
        if (leftrow[row] == 0 && lowerDiagonal[row + col] == 0 &&
upperDiagonal[n - 1 + col - row] == 0) {
          board[row][col] = 'Q';
          leftrow[row] = 1;
          lowerDiagonal[row + col] = 1;
          upperDiagonal[n - 1 + col - row] = 1;
          solve(col + 1, board, ans, leftrow, upperDiagonal,
lowerDiagonal, n);
          board[row][col] = '.';
          leftrow[row] = 0;
          lowerDiagonal[row + col] = 0;
          upperDiagonal[n - 1 + col - row] = 0;
        }
      }
    }

  public:
    vector < vector < string >> solveNQueens(int n) {
      vector < vector < string >> ans;
      vector < string > board(n);
      string s(n, '.');
      for (int i = 0; i < n; i++) {
        board[i] = s;
      }
      vector < int > leftrow(n, 0), upperDiagonal(2 * n - 1, 0),
lowerDiagonal(2 * n - 1, 0);
      solve(0, board, ans, leftrow, upperDiagonal, lowerDiagonal, n);
      return ans;
    }
};
int main() {
  int n = 4; // we are taking 4*4 grid and 4 queens
  Solution obj;
  vector < vector < string >> ans = obj.solveNQueens(n);
  for (int i = 0; i < ans.size(); i++) {
    cout << "Arrangement " << i + 1 << "\n";
    for (int j = 0; j < ans[0].size(); j++) {
      cout << ans[i][j];
      cout << endl;
    }
    cout << endl;
  }
  return 0;
}
```

Output:

```
Arrangement 1
..Q.
Q…
…Q
.Q..
Arrangement 2
.Q..
…Q
Q…
..Q.
```

Time Complexity: Exponential in nature since we are trying out all ways, to be precise it is O(N! * N).

Space Complexity: O(N)

Sudoku Solver

Problem Statement:
Given a 9×9 incomplete sudoku, solve it such that it becomes valid sudoku. Valid sudoku has the following properties.

1. All the rows should be filled with numbers(1 – 9) exactly once.
2. All the columns should be filled with numbers(1 – 9) exactly once.
3. Each 3×3 submatrix should be filled with numbers(1 – 9) exactly once.

Note: Character **'.'** indicates empty cell.

Example:

`Input:`

9	5	7	-	1	3	-	8	4
4	8	3	-	5	7	1	-	6
-	1	2	-	4	9	5	3	7
1	7	-	3	-	4	9	-	2
5	-	4	9	7	-	3	6	-
3	-	9	5	-	8	7	-	1
8	4	5	7	9	-	6	1	3
-	9	1	-	3	6	-	7	5
7	-	6	1	8	5	4	-	9

`Output:`

9	5	7	6	1	3	2	8	4
4	8	3	2	5	7	1	9	6
6	1	2	8	4	9	5	3	7
1	7	8	3	6	4	9	5	2
5	2	4	9	7	1	3	6	8
3	6	9	5	2	8	7	4	1
8	4	5	7	9	2	6	1	3
2	9	1	4	3	6	8	7	5
7	3	6	1	8	5	4	2	9

`Explanation:`

The empty cells are filled with the possible numbers. There can exist many such arrangements of numbers. The above solution is one of them. Let's see how we can fill the cells below.

Solution

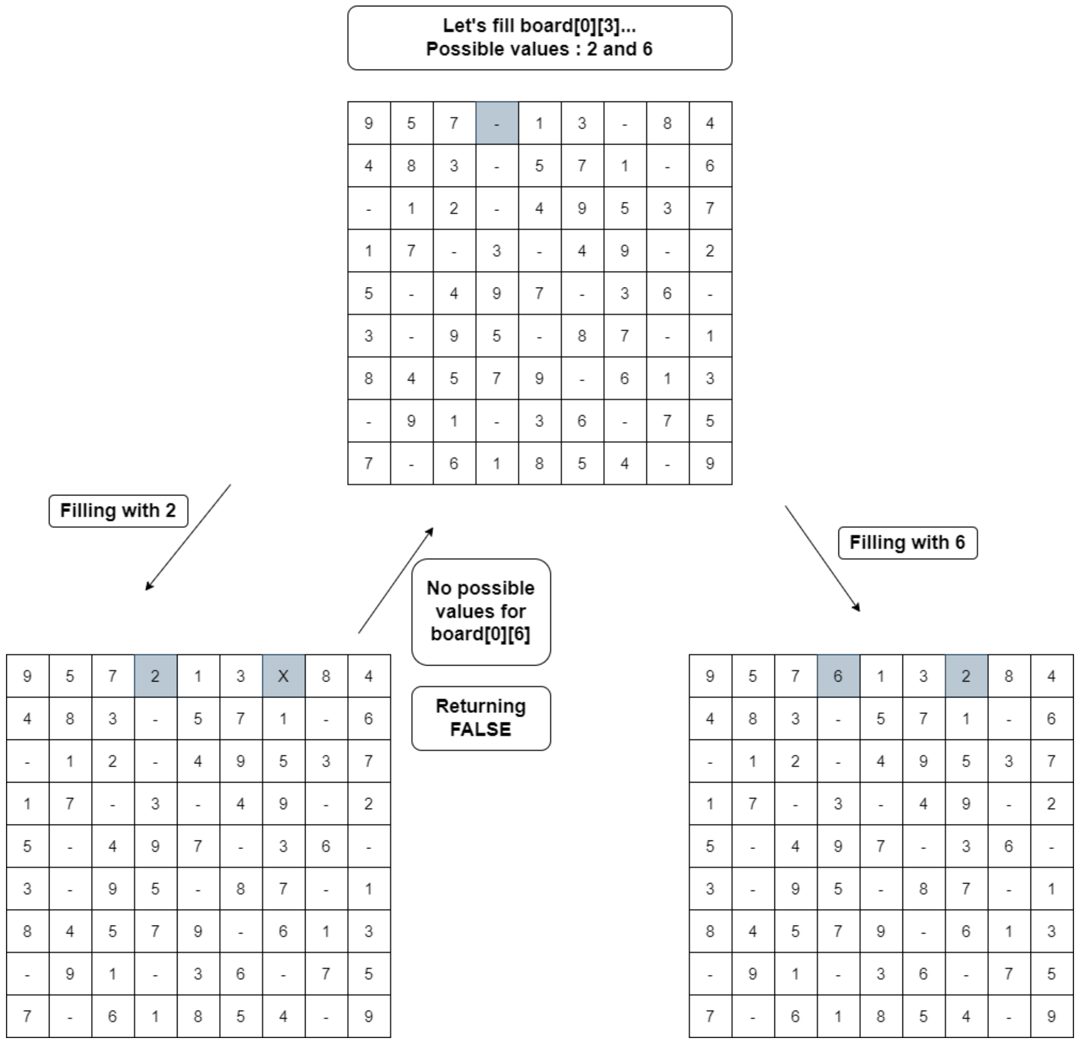
***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Intuition:

Since we have to fill the empty cells with available possible numbers and we can also have multiple solutions, the main intuition is to try every possible way of filling the empty cells. And the more correct way to try all possible solutions is to use recursion. In each call to the recursive function, we just try all the possible numbers for a particular cell and transfer the updated board to the next recursive call.

Approach:

- Let’s see the step by step approach. Our main recursive function(solve()) is going to just do a plain matrix traversal of the sudoku board. When we find an empty cell, we pause and try to put all available numbers(1 – 9) in that particular empty cell.
- We need another loop to do that. But wait, we forgot one thing – the board has to satisfy all the conditions, right? So, for that we have another function(isValid()) which will check whether the number we have inserted into that empty cell will not violate any conditions.
- If it is violating, we try with the next number. If it is not, we call the same function recursively, but this time with the updated state of the board. Now, as usual it tries to fill the remaining cells in the board in the same way.
- Now we’ll come to the returning values. If at any point we cannot insert any numbers from 1 – 9 in a particular cell, it means the current state of the board is wrong and we need to backtrack. An important point to follow is, we need to return false to let the parent function(which is called this function) know that we cannot fill this way. This will serve as a hint to that function, that it needs to try with the next possible number. Refer to the picture below.


Let's fill board[0][3]...
Possible values : 2 and 6
Filling with 2
No possible values for board[0][6]
Returning FALSE
Filling with 6

9	5	7	-	1	3	-	8	4
4	8	3	-	5	7	1	-	6
-	1	2	-	4	9	5	3	7
1	7	-	3	-	4	9	-	2
5	-	4	9	7	-	3	6	-
3	-	9	5	-	8	7	-	1
8	4	5	7	9	-	6	1	3
-	9	1	-	3	6	-	7	5
7	-	6	1	8	5	4	-	9

9	5	7	2	1	3	X	8	4
4	8	3	-	5	7	1	-	6
-	1	2	-	4	9	5	3	7
1	7	-	3	-	4	9	-	2
5	-	4	9	7	-	3	6	-
3	-	9	5	-	8	7	-	1
8	4	5	7	9	-	6	1	3
-	9	1	-	3	6	-	7	5
7	-	6	1	8	5	4	-	9

9	5	7	6	1	3	2	8	4
4	8	3	-	5	7	1	-	6
-	1	2	-	4	9	5	3	7
1	7	-	3	-	4	9	-	2
5	-	4	9	7	-	3	6	-
3	-	9	5	-	8	7	-	1
8	4	5	7	9	-	6	1	3
-	9	1	-	3	6	-	7	5
7	-	6	1	8	5	4	-	9

- If a recursive call returns true, we can assume that we found one possible way of filling and we simply do an **early return**.

Validating Board

- Now, let's see how we are validating the sudoku board. After determining a number for a cell(at i'th row, j'th col), we try to check the validity. As we know, a valid sudoku needs to satisfy 3 conditions, we can use three loops. But we can do within a single loop itself. Let's try to understand that.
- We loop from 0 to 8 and check the values – board[i][col](1st condition) and board[row][i](2nd condition), whether the number is already included. For the 3rd condition – the expression (3 * (row / 3) + i / 3) evaluates to the row numbers of that 3×3 submatrix and the expression (3 * (col / 3) + i % 3) evaluates to the column numbers.
- For eg, if row= 5 and col= 3, the cells visited are

	3	4	5
3	3	6	4
4	9	7	1
5	5	2	8

It covers all the cells in the sub-matrix.

Code:
```
#include <iostream>
#include <vector>
using namespace std;
bool isValid(vector < vector < char >> & board, int row, int col,
char c) {
  for (int i = 0; i < 9; i++) {
    if (board[i][col] == c)
      return false;

    if (board[row][i] == c)
      return false;

    if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
      return false;
  }
  return true;
}

bool solve(vector < vector < char >> & board) {
  for (int i = 0; i < board.size(); i++) {
    for (int j = 0; j < board[0].size(); j++) {
      if (board[i][j] == '.') {
        for (char c = '1'; c <= '9'; c++) {
          if (isValid(board, i, j, c)) {
            board[i][j] = c;

            if (solve(board))
              return true;
            else
              board[i][j] = '.';
          }
        }

        return false;
      }
    }
  }
  return true;
}
int main() {
    vector<vector<char>>board{
        {'9', '5', '7', '.', '1', '3', '.', '8', '4'},
        {'4', '8', '3', '.', '5', '7', '1', '.', '6'},
        {'.', '1', '2', '.', '4', '9', '5', '3', '7'},
        {'1', '7', '.', '3', '.', '4', '9', '.', '2'},
        {'5', '.', '4', '9', '7', '.', '3', '6', '.'},
        {'3', '.', '9', '5', '.', '8', '7', '.', '1'},
        {'8', '4', '5', '7', '9', '.', '6', '1', '3'},
        {'.', '9', '1', '.', '3', '6', '.', '7', '5'},
        {'7', '.', '6', '1', '8', '5', '4', '.', '9'}
    };

    solve(board);

    for(int i= 0; i< 9; i++){
        for(int j= 0; j< 9; j++)
            cout<<board[i][j]<<" ";
            cout<<"\n";
    }
    return 0;
}
```

Output:

```
9 5 7 6 1 3 2 8 4
4 8 3 2 5 7 1 9 6
6 1 2 8 4 9 5 3 7
1 7 8 3 6 4 9 5 2
```

```
5 2 4 9 7 1 3 6 8
3 6 9 5 2 8 7 4 1
8 4 5 7 9 2 6 1 3
2 9 1 4 3 6 8 7 5
7 3 6 1 8 5 4 2 9
```

Time Complexity: O(9(n ^ 2)), in the worst case, for each cell in the n2 board, we have 9 possible numbers.
Space Complexity: O(1), since we are refilling the given board itself, there is no extra space required, so constant space complexity.

M – Coloring Problem

Problem Statement: Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with the same color.
Examples:

```
Example 1:
Input:
N = 4
M = 3
E = 5
Edges[] = {
  (0, 1),
  (1, 2),
  (2, 3),
  (3, 0),
  (0, 2)
}
Output: 1
Explanation: It is possible to colour the given graph using 3
colours.
Example 2:
Input:
N = 3
M = 2
E = 3
Edges[] = {
  (0, 1),
  (1, 2),
  (0, 2)
}
Output: 0
```

Explanation: It is not possible to color.

Solution

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Backtracking

Approach:

Basically starting from vertex 0 color one by one the different vertices.

Base condition:

If I have colored all the N nodes return true.

Recursion:

Trying every color from 1 to m with the help of a for a loop.

Is safe function returns true if it is possible to color it with that color i.e none of the adjacent nodes have the same color.

In case this is an algorithm and follows a certain intuition, please mention it.

Color it with color i then call the recursive function for the next node if it returns true we will return true.

And If it is false then take off the color.

Now if we have tried out every color from 1 to m and it was not possible to color it with any of the m colors then return false.

Code:

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

using namespace std;
bool isSafe(int node, int color[], bool graph[101][101], int n, int
col) {
  for (int k = 0; k < n; k++) {
    if (k != node && graph[k][node] == 1 && color[k] == col) {
      return false;
    }
  }
  return true;
}
bool solve(int node, int color[], int m, int N, bool graph[101][101])
{
  if (node == N) {
    return true;
  }

  for (int i = 1; i <= m; i++) {
    if (isSafe(node, color, graph, N, i)) {
      color[node] = i;
      if (solve(node + 1, color, m, N, graph)) return true;
      color[node] = 0;
    }
  }
  return false;
}

//Function to determine if graph can be coloured with at most M
colours such
//that no two adjacent vertices of graph are coloured with same
colour.
bool graphColoring(bool graph[101][101], int m, int N) {
  int color[N] = {
    0
  };
  if (solve(0, color, m, N, graph)) return true;
  return false;
}

int main() {
  int N = 4;
  int m = 3;

  bool graph[101][101] = {
    (0, 1),
    (1, 2),
    (2, 3),
    (3, 0),
    (0, 2)
  };
  cout << graphColoring(graph, m, N);

}
```

Output: 1
Time Complexity: O(N^M) (n raised to m)
Space Complexity: O(N)

Rat in a Maze

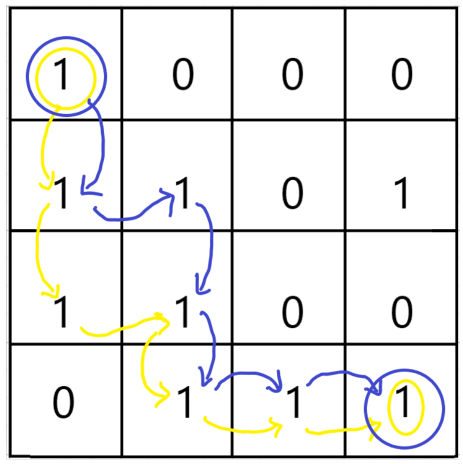
Rat in a Maze
Consider a rat placed at **(0, 0)** in a square matrix of order **N * N**. It has to reach the destination at **(N – 1, N – 1)**. Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are **‘U'(up)**, **‘D'(down)**, **‘L’ (left)**, **‘R’ (right)**. Value 0 at a cell in the matrix represents that it is blocked and the rat cannot move to it while value 1 at a cell in the matrix represents that rat can travel through it.
Note: In a path, no cell can be visited more than one time.

Print the answer in lexicographical(sorted) order

Examples:

Example 1:

```
Input:
N = 4
m[][] = {{1, 0, 0, 0},
         {1, 1, 0, 1},
         {1, 1, 0, 0},
         {0, 1, 1, 1}}

Output: DDRDRR DRDDRR

Explanation:
```



```
The rat can reach the destination at (3, 3) from (0, 0) by two paths
- DRDDRR and DDRDRR, when printed in sorted order we get DDRDRR
DRDDRR.
```

Example 2:

```
Input: N = 2
       m[][] = {{1, 0},
                {1, 0}}

Output:
 No path exists and the destination cell is blocked.
```

Solution

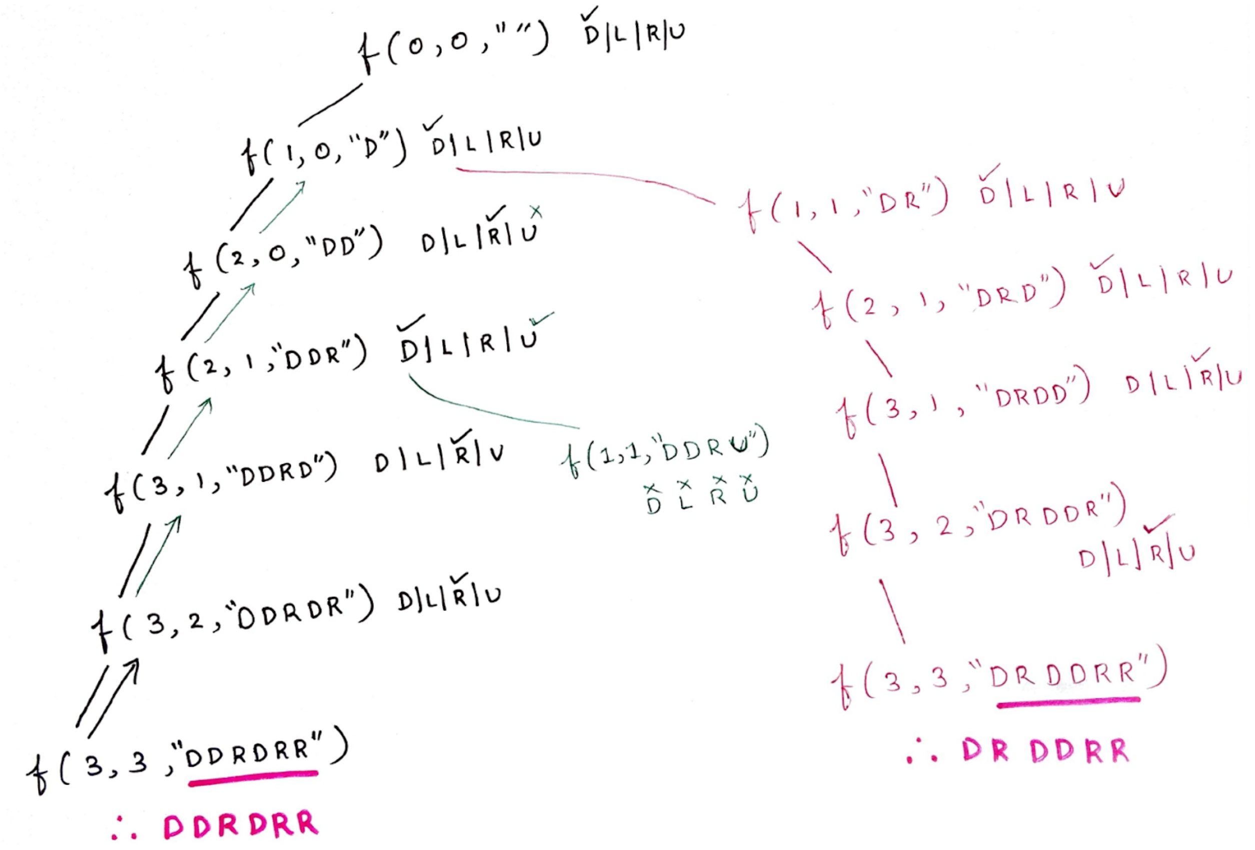
Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Recursion

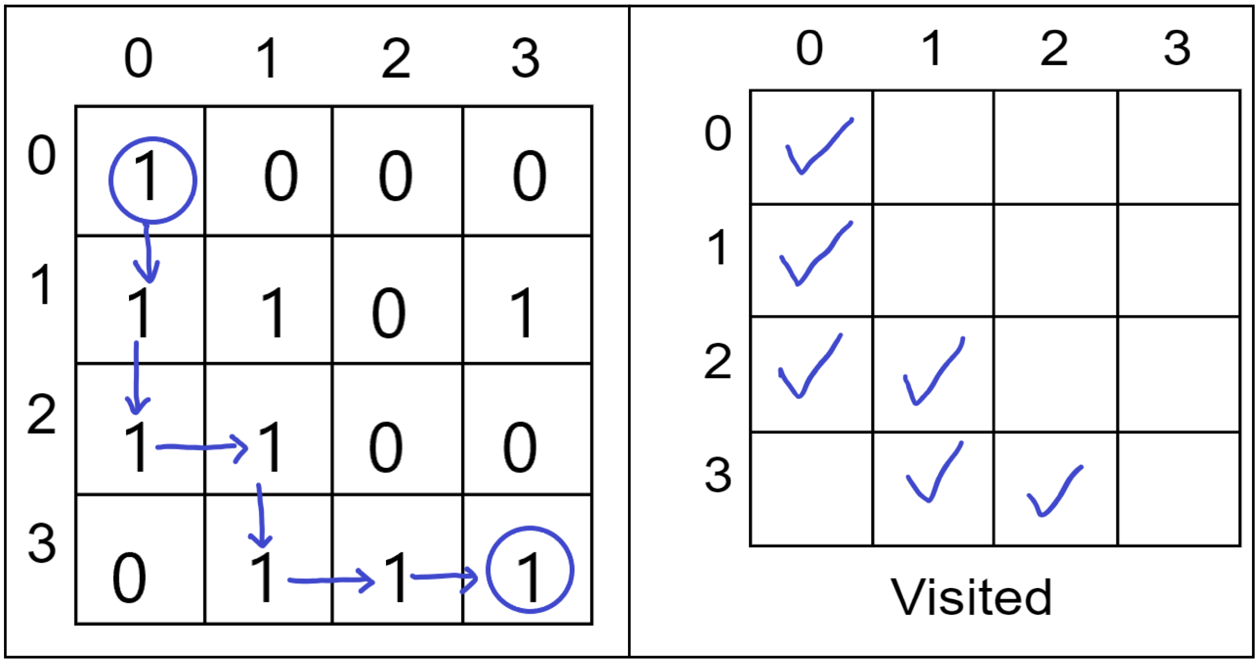
Intuition:
The best way to solve such problems is using recursion.
Approach:

- Start at the source(0,0) with an empty string and try every possible path i.e upwards**(U)**, downwards**(D)**, leftwards**(L)** and rightwards**(R)**.
- As the **answer** should be in lexicographical order so it's better to try the **directions** in lexicographical order i.e (D,L,R,U)
- Declare a 2D-array named visited because the question states that a single cell should be included only once in the path,so it's important to keep track of the visited cells in a particular path.
- If a cell is in path, mark it in the visited array.
- Also keep a check of the **"out of bound"** conditions while going in a particular direction in the matrix.
- Whenever you reach the destination**(n,n)** it's very important to get back as shown in the recursion tree.
- While getting back, keep on unmarking the visited array for the respective direction.Also check whether there is a different path possible while getting back and if yes, then mark that cell in the visited array.

Recursive tree:



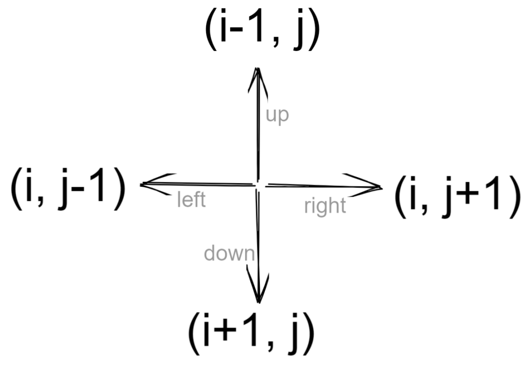
For “DDRDRR” :



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

using namespace std;

class Solution {
  void solve(int i, int j, vector < vector < int >> & a, int n,
vector < string > & ans, string move,
    vector < vector < int >> & vis) {
    if (i == n - 1 && j == n - 1) {
      ans.push_back(move);
      return;
    }

    // downward
    if (i + 1 < n && !vis[i + 1][j] && a[i + 1][j] == 1) {
      vis[i][j] = 1;
      solve(i + 1, j, a, n, ans, move + 'D', vis);
      vis[i][j] = 0;
    }
    // left
    if (j - 1 >= 0 && !vis[i][j - 1] && a[i][j - 1] == 1) {
      vis[i][j] = 1;
      solve(i, j - 1, a, n, ans, move + 'L', vis);
      vis[i][j] = 0;
    }

    // right
    if (j + 1 < n && !vis[i][j + 1] && a[i][j + 1] == 1) {
      vis[i][j] = 1;
      solve(i, j + 1, a, n, ans, move + 'R', vis);
      vis[i][j] = 0;
    }

    // upward
    if (i - 1 >= 0 && !vis[i - 1][j] && a[i - 1][j] == 1) {
      vis[i][j] = 1;
      solve(i - 1, j, a, n, ans, move + 'U', vis);
      vis[i][j] = 0;
    }

  }
  public:
    vector < string > findPath(vector < vector < int >> & m, int n) {
      vector < string > ans;
      vector < vector < int >> vis(n, vector < int > (n, 0));

      if (m[0][0] == 1) solve(0, 0, m, n, ans, "", vis);
      return ans;
    }
};

int main() {
  int n = 4;

   vector < vector < int >> m =
{{1,0,0,0},{1,1,0,1},{1,1,0,0},{0,1,1,1}};

  Solution obj;
  vector < string > result = obj.findPath(m, n);
  if (result.size() == 0)
    cout << -1;
  else
    for (int i = 0; i < result.size(); i++) cout << result[i] << " ";
  cout << endl;
  return 0;
}
```

Output:
DDRDRR DRDDRR
Time Complexity: O(4^(m*n)), because on every cell we need to try 4 different directions.
Space Complexity: O(m*n) ,Maximum Depth of the recursion tree(auxiliary space).

But, writing an individual code for every direction is a lengthy process therefore we truncate the 4 "if statements" into a single for loop using the following approach.



	D	L	R	U
di[ ]	+1	+0	+0	-1
dj[ ]	+0	-1	+1	+0

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

using namespace std;

class Solution {
  void solve(int i, int j, vector < vector < int >> & a, int n,
vector < string > & ans, string move,
    vector < vector < int >> & vis, int di[], int dj[]) {
    if (i == n - 1 && j == n - 1) {
      ans.push_back(move);
      return;
    }
    string dir = "DLRU";
    for (int ind = 0; ind < 4; ind++) {
      int nexti = i + di[ind];
      int nextj = j + dj[ind];
      if (nexti >= 0 && nextj >= 0 && nexti < n && nextj < n &&
!vis[nexti][nextj] && a[nexti][nextj] == 1) {
        vis[i][j] = 1;
        solve(nexti, nextj, a, n, ans, move + dir[ind], vis, di, dj);
        vis[i][j] = 0;
      }
    }
  }
  public:
    vector < string > findPath(vector < vector < int >> & m, int n) {
      vector < string > ans;
      vector < vector < int >> vis(n, vector < int > (n, 0));
      int di[] = {
        +1,
        0,
        0,
        -1
      };
      int dj[] = {
        0,
        -1,
        1,
        0
      };
      if (m[0][0] == 1) solve(0, 0, m, n, ans, "", vis, di, dj);
      return ans;
    }
};
int main() {
  int n = 4;
 vector < vector < int >> m =
{{1,0,0,0},{1,1,0,1},{1,1,0,0},{0,1,1,1}};

  Solution obj;
  vector < string > result = obj.findPath(m, n);
  if (result.size() == 0)
    cout << -1;
  else
    for (int i = 0; i < result.size(); i++) cout << result[i] << " ";
  cout << endl;

  return 0;
}
```

Output:

DDRDRR DRDDRR
Time Complexity: O(4^(m*n)), because on every cell we need to try 4 different directions.
Space Complexity: O(m*n) ,Maximum Depth of the recursion tree(auxiliary space).

Word Break Problem using Backtracking

- Difficulty Level : Hard
- Last Updated : 08 Nov, 2021

Given a valid sentence without any spaces between the words and a dictionary of valid English words, find all possible ways to break the sentence into individual dictionary words.

Example

```
Consider the following dictionary
{ i, like, sam, sung, samsung, mobile, ice,
  and, cream, icecream, man, go, mango}

Input: "ilikesamsungmobile"
Output: i like sam sung mobile
        i like samsung mobile

Input: "ilikeicecreamandmango"
Output: i like ice cream and man go
        i like ice cream and mango
        i like icecream and man go
        i like icecream and mango
```

Recommended Practice
Word Break – Part 2
Try It!
We have discussed a Dynamic Programming solution in the below post.
Dynamic Programming | Set 32 (Word Break Problem)
The Dynamic Programming solution only finds whether it is possible to break a word or not. Here we need to print all possible word breaks.
We start scanning the sentence from the left. As we find a valid word, we need to check whether the rest of the sentence can make valid words or not. Because in some situations the first found word from the left side can leave a remaining portion that is not further separable. So, in that case, we should come back and leave the currently found word and keep on searching for the next word. And this process is recursive because to find out whether the right portion is separable or not, we need the same logic. So we will use recursion and backtracking to solve this problem. To keep track of the found words we will use a stack. Whenever the

right portion of the string does not make valid words, we pop the top string from the stack and continue finding.

```
// A recursive program to print all possible
// partitions of a given string into dictionary
// words
#include <iostream>
using namespace std;

/* A utility function to check whether a word
  is present in dictionary or not.  An array of
  strings is used for dictionary.  Using array
  of strings for dictionary is definitely not
  a good idea. We have used for simplicity of
  the program*/
int dictionaryContains(string &word)
{
    string dictionary[] =
{"mobile","samsung","sam","sung",
                            "man","mango",
"icecream","and",

"go","i","love","ice","cream"};
    int n = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < n; i++)
        if (dictionary[i].compare(word) == 0)
            return true;
    return false;
}

// Prototype of wordBreakUtil
void wordBreakUtil(string str, int size, string
result);

// Prints all possible word breaks of given string
void wordBreak(string str)
{
    // Last argument is prefix
    wordBreakUtil(str, str.size(), "");
}
```

```
// Result store the current prefix with spaces
// between words
void wordBreakUtil(string str, int n, string result)
{
    //Process all prefixes one by one
    for (int i=1; i<=n; i++)
    {
        // Extract substring from 0 to i in prefix
        string prefix = str.substr(0, i);

        // If dictionary contains this prefix, then
        // we check for remaining string. Otherwise
        // we ignore this prefix (there is no else
for
        // this if) and try next
        if (dictionaryContains(prefix))
        {
            // If no more elements are there, print
it
            if (i == n)
            {
                // Add this element to previous
prefix
                result += prefix;
                cout << result << endl;
                return;
            }
            wordBreakUtil(str.substr(i, n-i), n-i,
                                result + prefix + "
");
        }
    }
}

//Driver Code
int main()
{
    // Function call
    cout << "First Test:\n";
    wordBreak("iloveicecreamandmango");

    cout << "\nSecond Test:\n";
    wordBreak("ilovesamsungmobile");
    return 0;
}
```

Output

```
First Test:
i love ice cream and man go
i love ice cream and mango
i love icecream and man go
i love icecream and mango

Second Test:
i love sam sung mobile
i love samsung mobile
```

Complexities:

- **Time Complexity**: O(2_n). Because there are 2_n combinations in The Worst Case.
- **Auxiliary Space**: O(n_2). Because of the Recursive Stack of wordBreakUtil(…) function in The Worst Case.

Where n is the length of the input string.