Striver’s SDE Sheet

Day 10: Recursion and Backtracking

Print Permutations - String

Find all permutation

Ans:

```
#include <bits/stdc++.h>
void func(string &s, int index, vector<string> &ans)
{
    if (index == s.length())
    {
        ans.push_back(s);
        return;
    }

    for (int i = index; i < s.length(); i++)
    {
        swap(s[i],s[index]);
        func(s, index + 1 , ans);
        swap(s[i],s[index]);

    }

    // a.push_back(s[index]);
    //     func(s, index + 1, a, ans);
    //     a.pop_back();
}
vector<string> findPermutations(string &s) {
    // Write your code here.
     vector<string> ans;
    func(s, 0, ans);
    return ans;
}
```

M-Coloring Problem

given an undirected graph and an integer M. The task is to 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. Here coloring of a graph means the assignment of colors to all vertices. Print 1 if it is possible to colour vertices and 0 otherwise.

Ans:

```
class Solution{

public:

    bool solve(vector<int> adj[], int s, int m, int n, vector<int> &color){

        if(s==n){

            return true;

        }


        for(int i=1; i<=m; i++){

            if(isPossible(adj, s, n, i, color)){

                color[s]=i;

                if(solve(adj, s+1, m, n, color)==true){

                    return true;

                }

                color[s]=0;

            }

        }

        return false;

    }




    bool isPossible(vector<int> adj[], int s, int n, int c, vector<int> &color){

        for(int v: adj[s]){

            if(color[v]==c){

                return false;

            }

        }

        return true;

    }


    bool graphColoring(bool graph[101][101], int m, int n) {

        vector<int> adj[n];

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

```
            for(int j=0; j<n; j++){
                if(graph[i][j]==1){
                    adj[i].push_back(j);
                }
            }
        }
        vector<int> color(n,0);
        return solve(adj, 0, m, n, color);
    }
};
```

Rat in a Maze Problem - I

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 rat cannot move to it while value 1 at a cell in the matrix represents that rat can be travel through it. Note: In a path, no cell can be visited more than one time. If the source cell is 0, the rat cannot move to any other cell.

Ans:

```
class Solution{

    public:

    void dfs(int row,int col,vector<vector<int>>&m,int
&n,map<pair<int,int>,int>&vis,string &temp,vector<string> &res){

        if(row==n-1&&col==n-1&&m[row][col]==1){

            res.push_back(temp);

            temp="";

            return;

        }

        if(row<0||row>=n||col<0||col>=n||m[row][col]==0||vis[{row,col}]){

            temp="";

            return;

        }

        vis[{row,col}]=1;

        vector<pair<int,int>>dir={{-1,0},{1,0},{0,-1},{0,1}};

        for(auto ele:dir){

            int new_row=row+ele.first,new_col=col+ele.second;

            string new_temp=temp;

            if(ele.first==-1){

                new_temp+='U';

            }

            if(ele.first==1){

                new_temp+='D';

            }

            if(ele.second==-1){

                new_temp+='L';

            }
```

```
            if(ele.second==1){
                new_temp+='R';
            }
            dfs(new_row,new_col,m,n,vis,new_temp,res);
        }
        vis[{row,col}]=0;
    }
    vector<string> findPath(vector<vector<int>> &m, int n) {
        // Your code goes here
        vector<string>res;
        string temp="";
        map<pair<int,int>,int>vis;
        dfs(0,0,m,n,vis,temp,res);
        return res;
    }
};
```

N-Queens

The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return *all distinct solutions to the n-queens puzzle*. You may return the answer in any order.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Ans:

We just have to check in diagonal and column if any queen already present then leave that row

Code:

```
class Solution {

public:


    bool isSafe(int row, int col, vector<string> &board, int n)
    {
        int x=row;
        int y=col;

        // check for same row
        while(y>=0)
        {
            if(board[x][y]=='Q')
            {
                return false;
            }
            y--;
        }

        x=row;
        y=col;
```

```
        // check for upper diagonaal
        while(x>=0 && y>=0)
        {
            if(board[x][y]=='Q')
            {
                return false;
            }
            x--;
            y--;
        }

        x=row;
        y=col;

        // check for lower diagonaal
        while(x<n && y>=0)
        {
            if(board[x][y]=='Q')
            {
                return false;
            }
            x++;
            y--;
        }

        return true;
    }

    void solve(int col, vector<vector<string>> &ans, vector<string> &board,
int n)
    {
        // base case
        if(col==n)
        {
            ans.push_back(board);
```

```
            return;
        }

        // solve 1 case and rest recursion will handle
        for(int row=0;row<n;row++)
        {
            if(isSafe(row, col, board, n))
            {
                // put queen on board
                board[row][col]='Q';
                solve(col+1, ans, board, n);

                // backtrack
                board[row][col]='.';
            }
        }
    }

    vector<vector<string>> solveNQueens(int n) {

        vector<string> board(n, string(n, '.'));
        vector<vector<string>> ans;

        solve(0, ans, board, n);

        return ans;
    }
};
```

Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:

1. Each of the digits 1-9 must occur exactly once in each row.
2. Each of the digits 1-9 must occur exactly once in each column.

3. Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

The '.' character indicates empty cells.

Ans:

Approach is same as n queen , check if valid and backtrack

Code:

```
class Solution {
private:
    bool isValid(vector<vector<char>>& board, int i, int j, char val){
        int sRow = i/3*3;
        int sCol = j/3*3;

        for(int k=0; k<9; k++){
            if(board[i][k] == val) return false;
            if(board[k][j] == val) return false;
            if(board[k/3 + sRow][k%3 + sCol] == val) return false;
        }
        return true;
    }
    bool solver(vector<vector<char>>& board, int row, int col){
        if(row == 9) return true;

        if(board[row][col] == '.'){
            for(char ch='1'; ch<='9'; ch++){
                if(isValid(board,row,col,ch)){
                    board[row][col] = ch;
                    if(col < 8){
                        if(solver(board,row,col+1)) return true;
                    }
                    else if(solver(board,row+1,0)) return true;
                    board[row][col] = '.';
                }
            }
```

```
        }
        else{
            if(col < 8){
                if(solver(board,row,col+1)) return true;
            }
            else if(solver(board,row+1,0)) return true;
        }
        return false;
    }
public:
    void solveSudoku(vector<vector<char>>& board) {
        solver(board,0,0);
    }
};
```