BACKTRACKING

DATA STRUCTURES AND ALGORITHMS - 1

BACKTRACKING : CLASSIC PROBLEMS

GROUP 14

GANESH SUNDHAR S - CB.EN.U4AIE22017
KARTHIGAI SELVAM - CB.EN.U4AIE22025
SHRUTHIKAA V - CB.EN.U4AIE22047
BHAVYA SAINATH U - CB.EN.U4AIE22055

PROBLEM STATEMENT

NQUEENS : The problem is to print all the possible ways to place n queens in a n x n board where all the queens are in a non-attacking position.

KNIGHT TOURS : The problem is to traverse a knight in a n x n board where each square has to be traversed once and only once.

MAZE CHALLENGE : The problem is to print all the possible ways of traversing a complete rectangular board of given size where the only moves are to move up, down, right or left.

SUDOKU SOLVER : The problem is to solve a sudoku puzzle. Each sudoku puzzle has a 9 x 9 board in which we place numbers from 1 to 9 in a order which is constrained and is specific to each puzzle.

ALL THESE WERE IMPLEMENTED USING ARRAYS AS IT IS EASIER TO REPRESENT THE GAME BOARD OF ALL THESE PROBLEMS USING ARRAYS

MAIN FUNCTION COMMON FOR ALL

- The main function can be generalised for all the problems taken.
- In the main function, we take the input from the user :

    1. Board Size (Nqueens, Knight tours, Maze Challenge)
    2. Board Values (Sudoku Solver)

- Then, we check if the inputs are correct according to our problem.
- ex : If the values entered in a sudoku puzzle are between 1 to 9 and the puzzle is not invalid.
- Then we call the function to solve the problem in each case.

NQUEENS

GET FUNCTION

```
public static void get(boolean[][] board,int row){
    if(row == board.length){
        display(board);
        System.out.println();
        count++;
        return;
    }
    for(int i=0;i
        if(check(board,row,i)){
            board[row][i] = true;
            get(board,row+1);
            board[row][i] = false;
        }
    }

}
```

NQUEENS

CHECK FUNCTION

```
private static boolean check(boolean[][] board, int row, int col) {
    for(int i=0;i<board.length;i++){
        if(board[i][col] == true){
            return false;
        }
    }
    int lmin = Math.min(row,col),rmin = Math.min(row,board.length-col-1);
    int r = row,c = col;
```

```
for(int j=0;j<lmin;j++){
    row--;
    col--;
    if(board[row][col]){
        return false;
    }
}

for(int j=0;j<rmin;j++){
    r--;
    c++;
    if(board[r][c]){
        return false;
    }
}
return true;
```

NQUEENS

DISPLAY FUNCTION

```
public static void display(boolean[][] board) {
    for(boolean[] tem : board){
      for(boolean temp : tem){
        if(temp == true){
          System.out.print("Q ");
        }
        else{
          System.out.print("X ");
        }
      }
      System.out.println();
    }
}
```

DEMONSTRATION

KNIGHT TOURS

SOLVE FUNCTION

```
public static void solve(int[][] board, int a, int b,int step) {
    int n=board.length;
    int[][] values = {{2,2,1,1,-2,-2,-1,-1},{1,-1,2,-2,1,-1,2,-2}};
    board[a][b] = step;
    if(step>=n*n){
        board[a][b] = step;
        System.out.println("The way "+(count+1)+" is : ");
        display(board);
        System.out.println();
        count++;
    }
```

```
for(int i=0;i<8;i++){
    int a1 = a+values[0][i];
    int b1 = b+values[1][i];
    if(a1=0 && b1=0 && board[a1][b1] == 0){
        solve(board,a1,b1,step+1);
    }
}
board[a][b] = 0;
```

KNIGHT TOURS

DISPLAY FUNCTION

```
public static void display(int[][] board){
    for(int[] row : board){
        System.out.println(Arrays.toString(row));
    }
}
```

DEMONSTRATION

MAZE CHALLENGE

CALCULATE FUNCTION

```
static void calculate(int a,int b,boolean[][] array,String output,int x,int y,int[][] path,int step){
    int c=x,d=y;
    boolean[][] temp = array;
    if(c==(a-1) && d==(b-1)){
        System.out.println("Path "+(count+1)+" :");
        for(int[] arr : path  ){
            System.out.println(Arrays.toString(arr));
        }
        System.out.println("Path : "+output+"\n");
        count++;
        return;
    }
```

MAZE CHALLENGE

CALCULATE FUNCTION

```
if(c<a-1){
   c++;
   if(array[c][d] != false){
      array[c][d] = false;
      path[c][d] = step;
      calculate(a,b,array,output+"D",c,d,path,step+1);
      path[c][d] = 0;
      array[c][d] = true;
   }
}
```

MAZE CHALLENGE

CALCULATE FUNCTION

```
c=x;
d=y;
if(d<b-1){;
   d++;
   if(array[c][d] != false){
      array[c][d] = false;
      path[c][d] = step;
      calculate(a,b,array,output+"R",c,d,path,step+1);
      path[c][d] = 0;
      array[c][d] = true;
   }
}
```

MAZE CHALLENGE

CALCULATE FUNCTION

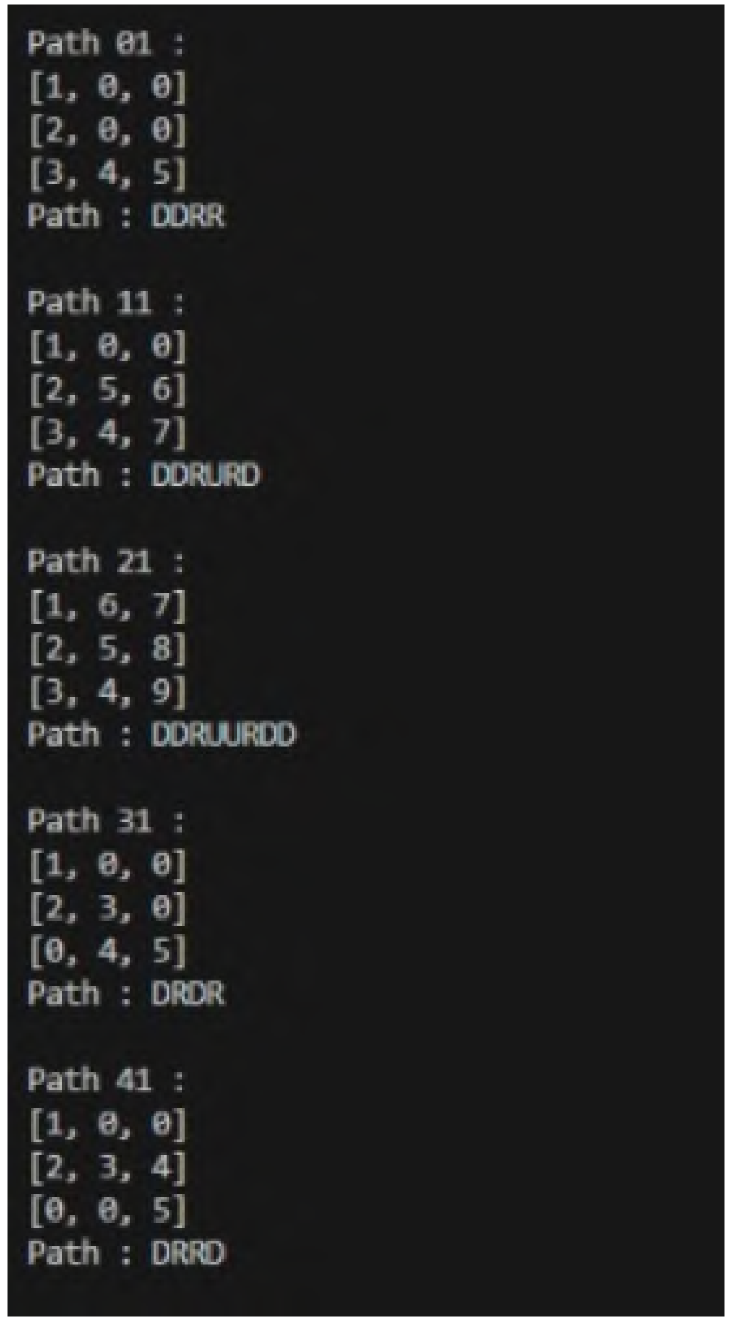
```
c=x;
d=y;
if(d>0){
   d--;
   if(array[c][d] == true){
      array[c][d] = false;
      path[c][d] = step;
      calculate(a,b,array,output+"L",c,d,path,step+1);
      path[c][d] = 0;
      array[c][d] = true;
   }
}
```

MAZE CHALLENGE

CALCULATE FUNCTION

```
c=x;
d=y;
if(c>0){
   c--;
   if(array[c][d] == true){
      array[c][d] = false;
      path[c][d] = step;
      calculate(a,b,array,output+"U",c,d,path,step+1);
      path[c][d] = 0;
      array[c][d] = true;
   }
}
```

DEMONSTRATION



```
Path 01 :
[1, 0, 0]
[2, 0, 0]
[3, 4, 5]
Path : DDRR

Path 11 :
[1, 0, 0]
[2, 5, 6]
[3, 4, 7]
Path : DDRURD

Path 21 :
[1, 6, 7]
[2, 5, 8]
[3, 4, 9]
Path : DDRUURDD

Path 31 :
[1, 0, 0]
[2, 3, 0]
[0, 4, 5]
Path : DRDR

Path 41 :
[1, 0, 0]
[2, 3, 4]
[0, 0, 5]
Path : DRRD
```

SUDOKU SOLVER (JAVA)

SOLVE FUNCTION

```
public static void solve(int[][] board, int a, int b) {

    int i,j,k=0;
    for(i=0;i<9;i++){
        for(j=0;j<9;j++){
            if(board[i][j] != 0){
                k++;
            }
        }
    }
```

SUDOKU SOLVER (JAVA)

SOLVE FUNCTION

```
if(k==81){
    System.out.println("\nThe solution is : ");
    for(int[] row : board){
        System.out.println(Arrays.toString(row));
    }
    System.out.println("This program will exit in a moment...");
    try{
    Thread.sleep(50000);
    }
    catch(Exception e){}
    System.exit(0);
}
```

SUDOKU SOLVER (JAVA)

SOLVE FUNCTION

```
if(board[a][b] != 0){
   if(b<8){
      solve(board,a,b+1);
      return;
   }
   else{
      solve(board,a+1,0);
      return;
   }
}
```

SUDOKU SOLVER (JAVA)

SOLVE FUNCTION

```
if(board[a][b] == 0){
    for(i=1;i<=9;i++){
        if(check(board,a,b,i)){
            board[a][b] = i;
            if(b<8){
                solve(board,a,b+1);
            }
            else{
                solve(board,a+1,0);
            }
            board[a][b] = 0;
        }
    }
}
```

SUDOKU SOLVER (JAVA)

CHECK FUNCTION

```
public static boolean check(int[][] board, int a, int b, int val) {

    int i,j;
    for(i=0;i<9;i++){
        if(board[a][i] == val || board[i][b] == val){
            return false;
        }
    }
    int c= (a/3) + 1;
    int d = (b/3) + 1;
    for(i=(c-1)*3 ; i< c*3 ;i++){
        for(j=(d-1)*3 ; j
            if(board[i][j] == val){
                return false;
            }
        }
    }

    return true;
}
```

DEMONSTRATION

SUDOKU SOLVER (C)

The **main** function initializes the Sudoku board, reads input from the user, and calls the **solve** function to find the solution. if user enters other than numbers it will show inavlid. if we enter numbers greater than 9 it will show invalid.

```
#include <stdio.h>
#include <stdlib.h>
void solve(int board[9][9], int a, int b);
int check(int board[9][9], int a, int b, int val);
int main() {
  int board[9][9];
   printf("Enter the Sudoku puzzle (9x9 grid):\n");
 for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
       int num;
   if (scanf("%d", &num) != 1) {
      printf("Invalid input. Please enter numbers only.\n");
       return 1;
 }                    // Validate the input number
if (num < 0 || num > 9) {
  printf("Invalid input. Please enter numbers from 0 to 9 only.\n");
    return 1;
 }
   board[i][j] = num;
 }
  }
  solve(board, 0, 0);
  return 0;
}
```

At each cell the code checks if the value is not equal to 0. If the cell is filled (contains a nonzero value), the **if** condition is true, and the variable **k** is incremented by 1.If all 81 cells are filled, it means the Sudoku is solved. The count (k) is compared against 81 to check if the Sudoku is already solved or if there are still empty cells that need to be filled. The first part of the solve function counts the number of filled cells in the Sudoku board by iterating through each cell. If all cells are filled (k == 81), it means the Sudoku is solved, and the solution is printed.

```
void solve(int board[9][9], int a, int b) {
  int i, j, k = 0;
  for (i = 0; i < 9; i++) {
    for (j = 0; j < 9; j++) {
    if (board[i][j] != 0) {
     k++;
 }
    }
        }

  if (k == 81) {
    printf("\nThe solution is :\n");
  for (i = 0; i < 9; i++) {
 for (j = 0; j < 9; j++) {
   printf("%d ", board[i][j]);
     }
printf("\n");
   }
exit(0);
}
```

If the current cell (board[a][b]) is already filled (not 0), the function moves to the next cell by recursively calling solve with updated position. It moves row-wise from left to right and top to bottom.

If the current cell is empty (board[a][b] == 0), the function tries to fill it with values 1 to 9 by calling the check function to validate the value. If a value is valid, it is assigned to the current cell, and solve is called recursively for the next cell.

```
  if (board[a][b] != 0) {
   if (b < 8) {
  solve(board, a, b + 1);
return;
 }
 else {
   solve(board, a + 1, 0);
     return;
  }
 }
```

```
if (board[a][b] == 0) {
    for (i = 1; i <= 9; i++) {
       if (check(board, a, b, i)) {
        board[a][b] = i;
     if (b < 8) {
  solve(board, a, b + 1);
          }
            else {
              solve(board, a + 1, 0);
    }
      board[a][b] = 0;
  }
    }
       }
          }
```

The **check** function is responsible for validating whether a given value (**val**) can be placed at a specific cell in the Sudoku board without violating the rules of Sudoku.

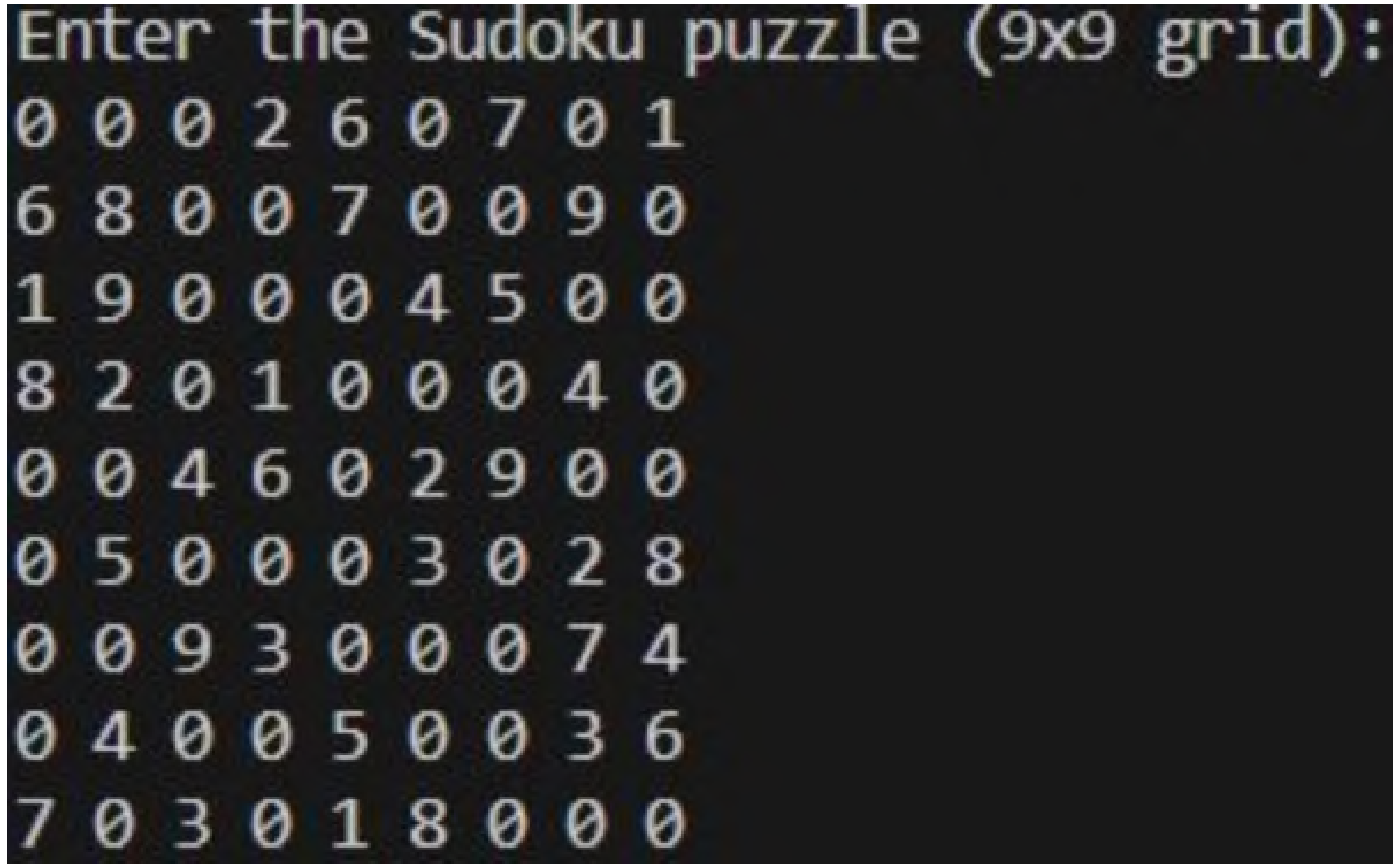
The first for loop iterates through the row a and the column b to check if the value val already exists in the same row or column. If a matching value is found, it means the value cannot be placed at the current cell, so the function returns 0 (not valid).

The another for loops iterate through the cells within the identified subgrid. The starting indices are calculated by subtracting 1 from c and d and multiplying them by 3. The loops then check if the value val already exists in any cell within the subgrid. If a matching value is found, the function returns 0 (not valid).

If the value val is not found in the row, column, or subgrid, the function returns 1, indicating that it is valid to place val at the current cell.

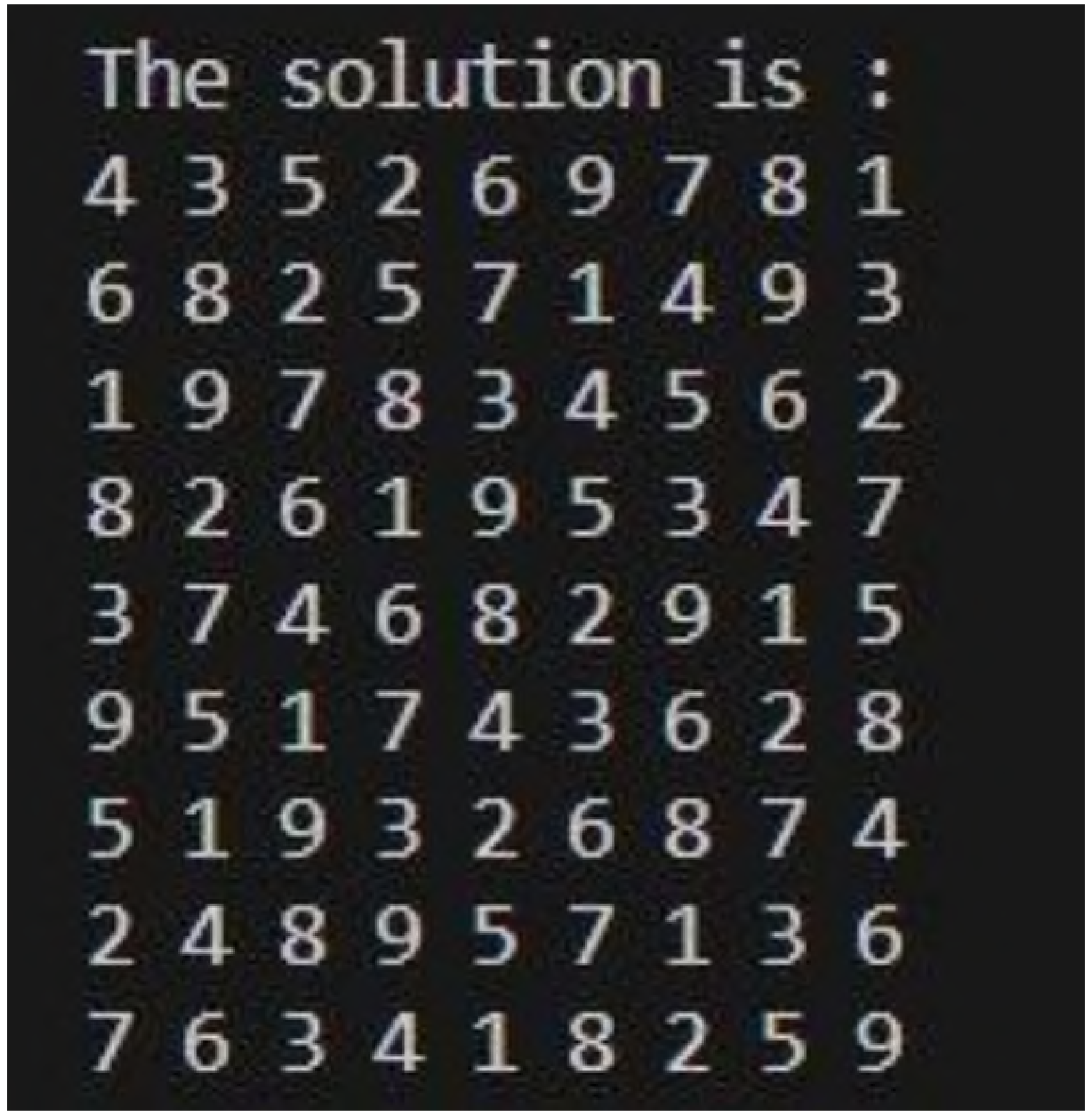
```
int check(int board[9][9], int a, int b, int val) {
  int i, j;
 for (i = 0; i < 9; i++) {
    if (board[a][i] == val || board[i][b] == val) {
    return 0;
   }
   }

int c = (a / 3) + 1;   int d = (b / 3) + 1;
    for (i = (c - 1) * 3; i < c * 3; i++) {
    for (j = (d - 1) * 3; j < d * 3; j++) {
        if (board[i][j] == val) {
      return 0;
   }
      }
        }
    return 1;
}
```

DEMONSTRATION



```
Enter the Sudoku puzzle (9x9 grid):
0 0 0 2 6 0 7 0 1
6 8 0 0 7 0 0 9 0
1 9 0 0 0 4 5 0 0
8 2 0 1 0 0 0 4 0
0 0 4 6 0 2 9 0 0
0 5 0 0 0 3 0 2 8
0 0 9 3 0 0 0 7 4
0 4 0 0 5 0 0 3 6
7 0 3 0 1 8 0 0 0
```

output:



```
The solution is :
4 3 5 2 6 9 7 8 1
6 8 2 5 7 1 4 9 3
1 9 7 8 3 4 5 6 2
8 2 6 1 9 5 3 4 7
3 7 4 6 8 2 9 1 5
9 5 1 7 4 3 6 2 8
5 1 9 3 2 6 8 7 4
2 4 8 9 5 7 1 3 6
7 6 3 4 1 8 2 5 9
```

THANK YOU

BACKTRACKING