INTRODUCTION

**Data structures** in C are used to store data in a computer in an organized form Data structures are inevitable part of programs. Computer programs frequently process data so we require efficient ways in which we can access or manipulate data. Some applications may require modification of data frequently and in others new data is constantly added or deleted. So we need efficient ways of accessing data so as to act on it and build efficient applications. In C language, Different types of data structures are; Array, Stack, Queue, Linked List, and Tree. For the maze problem, we make use of array, which is collection of elements of same data type referenced under a single name.

A Maze is a network of paths and hedges designed as a puzzle through which one has to find a way. We can write a computer program for getting through a maze. There are mainly three approaches to get through the maze – Dynamic Approaching, Backtracking, Greedy Approach. Here, we solve the problem using backtracking. Backtracking is a general [algorithm](https://en.wikipedia.org/wiki/Algorithm) for finding all (or some) solutions to some [computational problems](https://en.wikipedia.org/wiki/Computational_problem), notably [constraint satisfaction problems](https://en.wikipedia.org/wiki/Constraint_satisfaction_problem), that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution . Backtracking is an important tool for solving constraints satisfaction problems such as crosswords, verbal arithmetic, Sudoku and many other puzzles. The reason behind using backtracking is, that there are predefined constants. it also avoids undesired computations. Hence, it facilitates some ease for the programmer to choose the best solutions.

DESIGN

ALGORITHM:   
NAIVE ALGORITHM

The Naive Algorithm is to generate all paths from source to destination and one by one check if the generated path satisfies the constraints.

while there are untried paths

{

Generate the next paths

If this path has all blocks as 1

{

Print this path

}

}

BACKTRACKING ALGORITHM

if destination is reached

print the solution matrix

else

{

1. Mark current cell in solution matrix as 1
2. Move forward in the horizontal direction and recursively checks if this move leads to a solution.
3. If the move chosen in the above step doesn’t lead to a solution, then move down and check if this move leads to a solution.
4. If none of the above solutions works, then unmark this cell as 0 (BACKTRACK) and return false

}

ALGORITHM TO COUNT NUMBER OF PATHS:

We can recursively compute grid[i][j] using below

formula and finally return grid[R-1][C-1]

// If current cell is a blockage

if (maze[i][j] == -1)

maze[i][j] = -1; // Do not change

// If we can reach maze[i][j] from maze[i-1][j]

// then increment count.

else if (maze[i-1][j] > 0)

maze[i][j] = (maze[i][j] + maze[i-1][j]);

// If we can reach maze[i][j] from maze[i][j-1]

// then increment count.

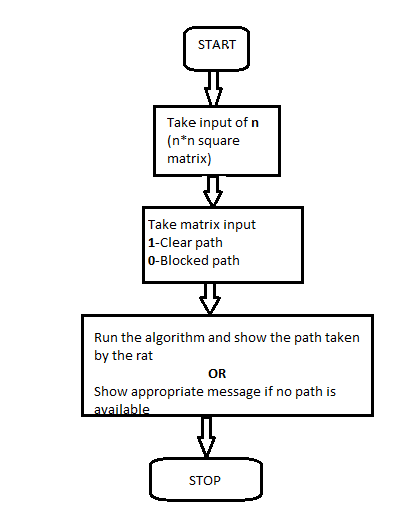
else if (maze[i][j-1] > 0)

maze[i][j] = (maze[i][j] + maze[i][j-1]);

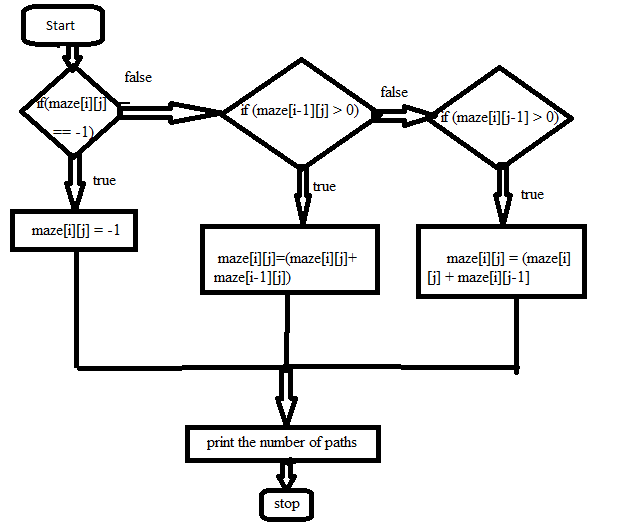
Thus, the summary is that we try to move to the other cell (down, right, up, and left) from the current cell and if no movement is possible, then just come back and change the direction of the path to another cell.

FLOWCHART

FLOWCHART FOR FINDING A PATH:



FLOWCHART FOR COUNTING THE PATHS:



IMPLEMENTATION

#include <stdio.h>

#include <conio.h>

#define SIZE 4

#define R 4

#define C 4

//the maze problem

int maze[SIZE][SIZE] = {

{0, 0, 0, 0},

{0, -1, 0, 0},

{-1, 0, 0, 0},

{0, 0, 0, 0}};

// Returns count of possible paths in a maze[R][C]

// from (0,0) to (R-1,C-1)

intcountPaths(int maze[R][C])

{

intI,j;

// If the initial cell is blocked, there is no

// way of moving anywhere

if (maze[0][0]==-1)

return 0;

// Initializing the leftmost column

for (i=0; i<R; i++)

{

if (maze[i][0] == 0)

maze[i][0] = 1;

// If we encounter a blocked cell in leftmost

// row, there is no way of visiting any cell

// directly below it.

else

break;

}

// Similarly initialize the topmost row

for (i=1; i<C; i++)

{

if (maze[0][i] == 0)

maze[0][i] = 1;

// If we encounter a blocked cell in bottommost

// row, there is no way of visiting any cell

// directly below it.

else

break;

}

// The only difference is that if a cell is -1,

// simply ignore it else recursively compute

// count value maze[i][j]

for (i=1; i<R; i++)

{

for (j=1; j<C; j++)

{

// If blockage is found, ignore this cell

if (maze[i][j] == -1)

continue;

// If we can reach maze[i][j] from maze[i-1][j]

// then increment count.

if (maze[i-1][j] > 0)

maze[i][j] = (maze[i][j] + maze[i-1][j]);

// If we can reach maze[i][j] from maze[i][j-1]

// then increment count.

if (maze[i][j-1] > 0)

maze[i][j] = (maze[i][j] + maze[i][j-1]);

}

}

// If the final cell is blocked, output 0, otherwise

// the answer

return (maze[R-1][C-1] > 0)? maze[R-1][C-1] : 0;

}

//matrix to store the solution

int solution[SIZE][SIZE];

//function to print the solution matrix

void printsolution()

{

inti,j;

for(i=0;i<SIZE;i++)

{

for(j=0;j<SIZE;j++)

{

printf("%d\t",solution[i][j]);

}

printf("\n\n");

}

getch();

}

//function to solve the maze

//using backtracking

intsolvemaze(int r, int c)

{

//if destination is reached, maze is solved

//destination is the last cell(maze[SIZE-1][SIZE-1])

if((r==SIZE-1) && (c==SIZE-1))

{

solution[r][c] = 1;

return 1;

}

//checking if we can visit in this cell or not

//the indices of the cell must be in (0,SIZE-1)

//and solution[r][c] == 0 is making sure that the cell is not already visited

//maze[r][c] == 0 is making sure that the cell is not blocked

if(r>=0 && c>=0 && r<SIZE && c<SIZE && solution[r][c] == 0 && maze[r][c] == 0)

{

//if safe to visit then visit the cell

solution[r][c] = 1;

//going down

if(solvemaze(r+1, c))

return 1;

//going right

if(solvemaze(r, c+1))

return 1;

//going up

if(solvemaze(r-1, c))

return 1;

//going left

if(solvemaze(r, c-1))

return 1;

//backtracking

solution[r][c] = 0;

return 0;

}

return 0;

}

intmain()

{

//making all elements of the solution matrix 0

inti,j,a;

for(i=0; i<SIZE; i++)

{

for(j=0; j<SIZE; j++)

{

solution[i][j] = 0;

}

}

if (solvemaze(0,0))

printsolution();

else

printf("No solution\n");

a= countPaths(maze);

printf("THE NUMBER OF PATHS ARE : %d\n",a);

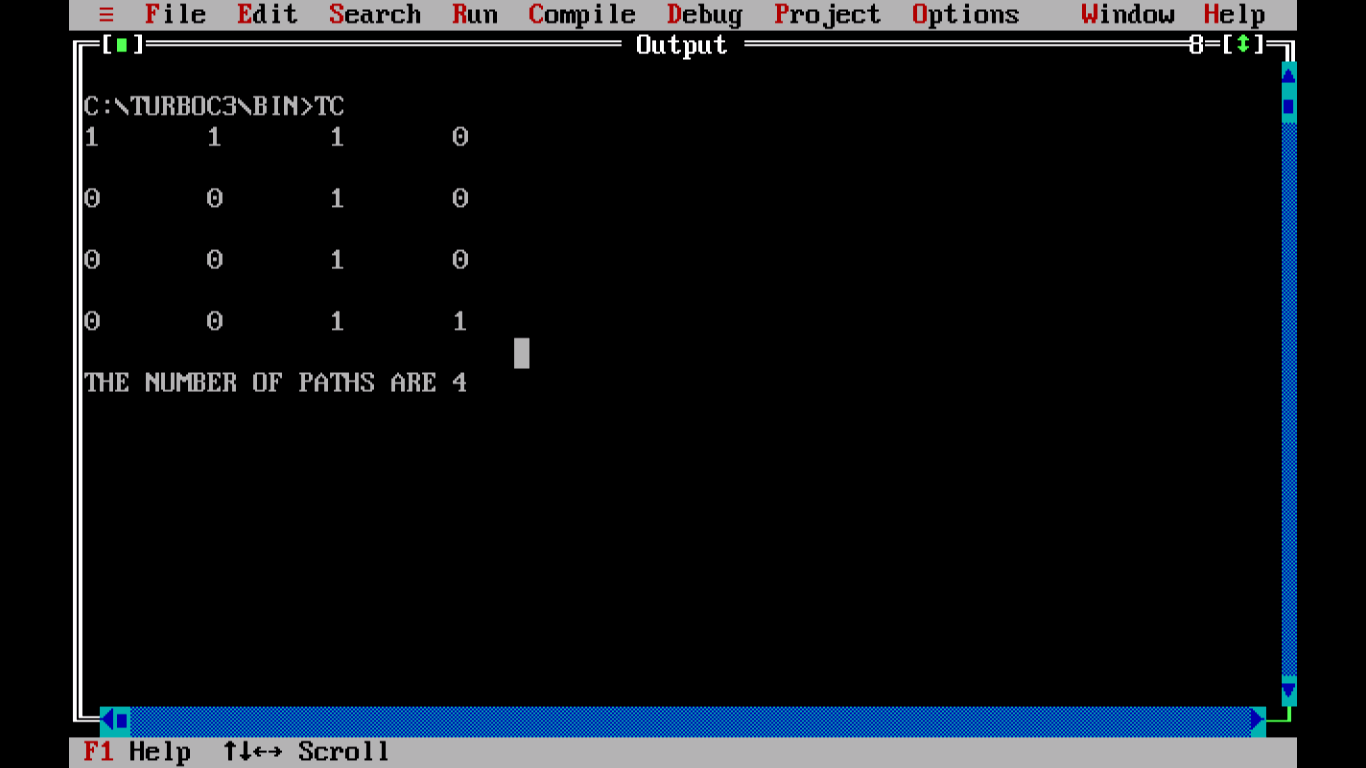
getch();

return 0;

}

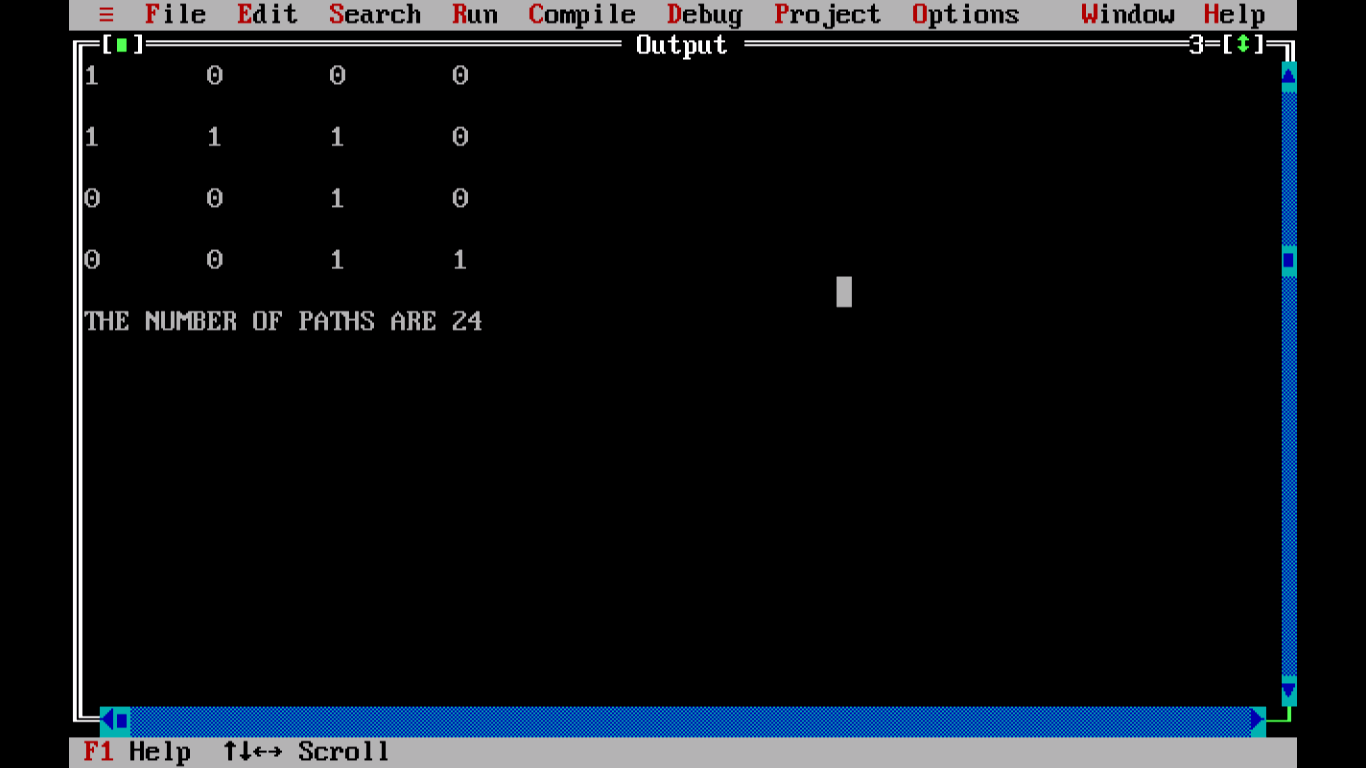
OUTPUT

Run 1:



The number of paths in this case is 4.

Run 2:



The number of paths in this case will be 24.

CONCLUSION

In the project, we solve the maze problem given by using backtracking approach, and also, we print the number of possible paths from the source to the destination of the maze. Including to this, we understand the functionality of recursive loops.

# FUTURE ENHANCEMENTS:

We can print all possible paths from source to destination.We can also print the shortest and longest path as well.

REFERENCES

* <https://www.codesdope.com/blog/article/backtracking-to-solve-a-rat-in-a-maze-c-java-pytho/>
* <https://www.geeksforgeeks.org/count-number-ways-reach-destination-maze/amp/>