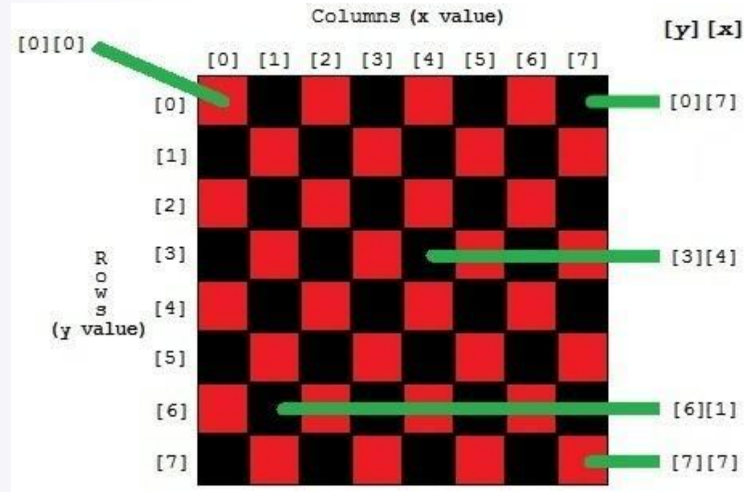
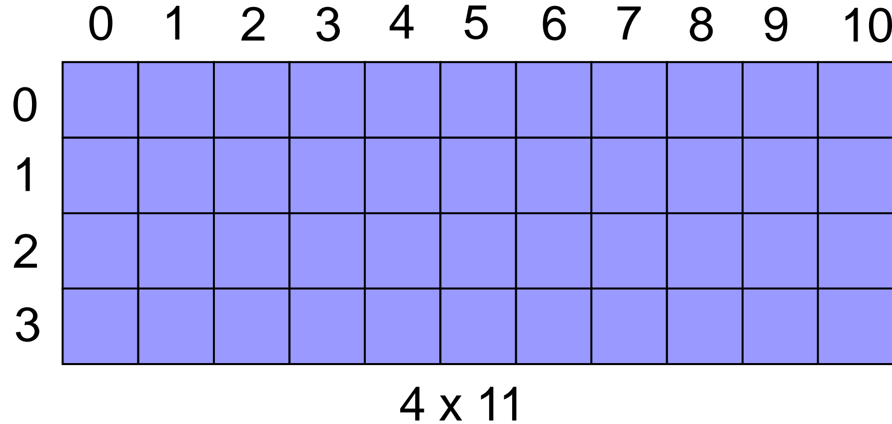
Two Dimensional Arrays



Two-dimensional Arrays

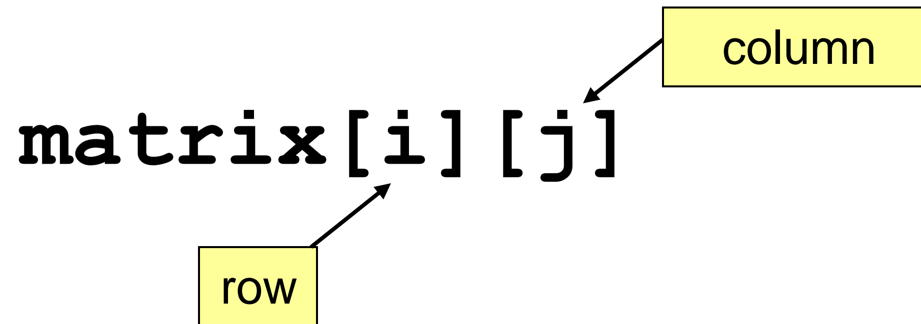


- Declaration:

```
int matrix[4][11];
```

Element Access

- Access the j-th element (column) of the i-th array (row)



Example: Matrix Addition

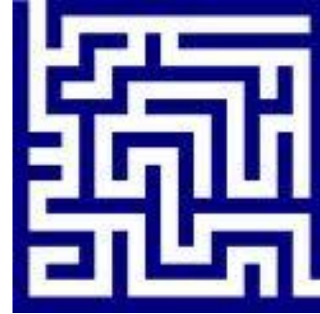
```
#define ROWS 3
#define COLS 4

int main()
{
    int a[ROWS][COLS] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    int b[ROWS][COLS] = {{1,1,1,1},{2,2,2,2},{3,3,3,3}};
    int c[ROWS][COLS];

    int i = 0, j = 0;

    for (i = 0; i < ROWS; ++i)
        for (j = 0; j < COLS; ++j)
            c[i][j] = a[i][j] + b[i][j];

    /* print c */
    ...

    return 0;
}
```

Exercise

- Calculate a matrix transpose:
  - define two int matrices **A** and **B** of size 5x5
  - initialize **A** with any values you want.
  - compute $\mathbf{A}^T$, the matrix $B[i]$ transpose of **A**, in to **B**:

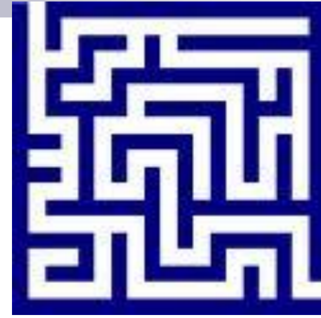
    B[i][j] = A[j][i]
  - Print **B**.

SOLVE A MAZE



- Our maze is represented by a rectangle divided into small squares = a matrix
- It has:
  - Starting point
  - End point
  - Walls
  - Empty cells that we can travel through

SOLVE A MAZE



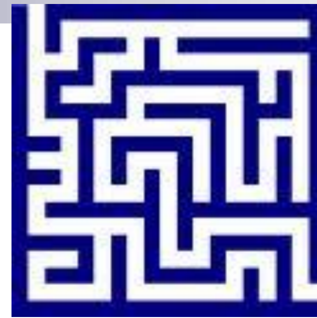
How do we solve a maze?

- We need to find a path from our current location to the end point.

Recursion:

- Find a path from a neighbor cell to the end point
- Add the current cell in the beginning of the path.

SOLVE A MAZE



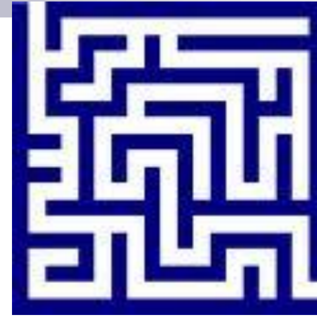
What happens if no path from any of our neighbors exists?

Return to the cell we arrived from.

Stopping criteria:

- Reach the End Point➔ **DONE**
- Hit a wall or an already visited cell ➔ **Reverse**

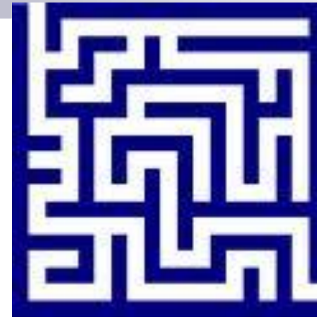
SOLVE A MAZE



Algorithm:

- When you reach a cell:
  - Try walking down / right / up / left in some order until:
  - **Success** = Reach END
  - **Failure** = Checked all possible paths

SOLVE A MAZE



Reverse when you hit:

- A Wall
- A border
- An already visited cell (not necessarily in the path)

The function ***solve*** receives:

- maze (2d matrix)
- (row, col) of current cell
- step (cell num on the path)

```
int solve(int maze[][WIDTH], int h, int w, int step)
{
    // check borders
    if ( (h < 0)  || (h >= HEIGHT) || (w < 0) || (w >= WIDTH) )
        return FALSE;
    switch (maze[h][w])
    {
        case END:
            return TRUE;
        case EMPTY:
            maze[h][w] = step;
            // the actual walk along the path
            if (solve(maze, h + 1, w, step + 1) || solve(maze, h, w + 1, step + 1) ||
                solve(maze, h - 1, w, step + 1) || solve(maze, h, w - 1, step + 1))
                    return TRUE;
            maze[h][w] = VISITED;       // Failed
        default:                        // PATH or VISITED or failed EMPTY
            return FALSE;
    }
}
```

SOLVE A MAZE – Main

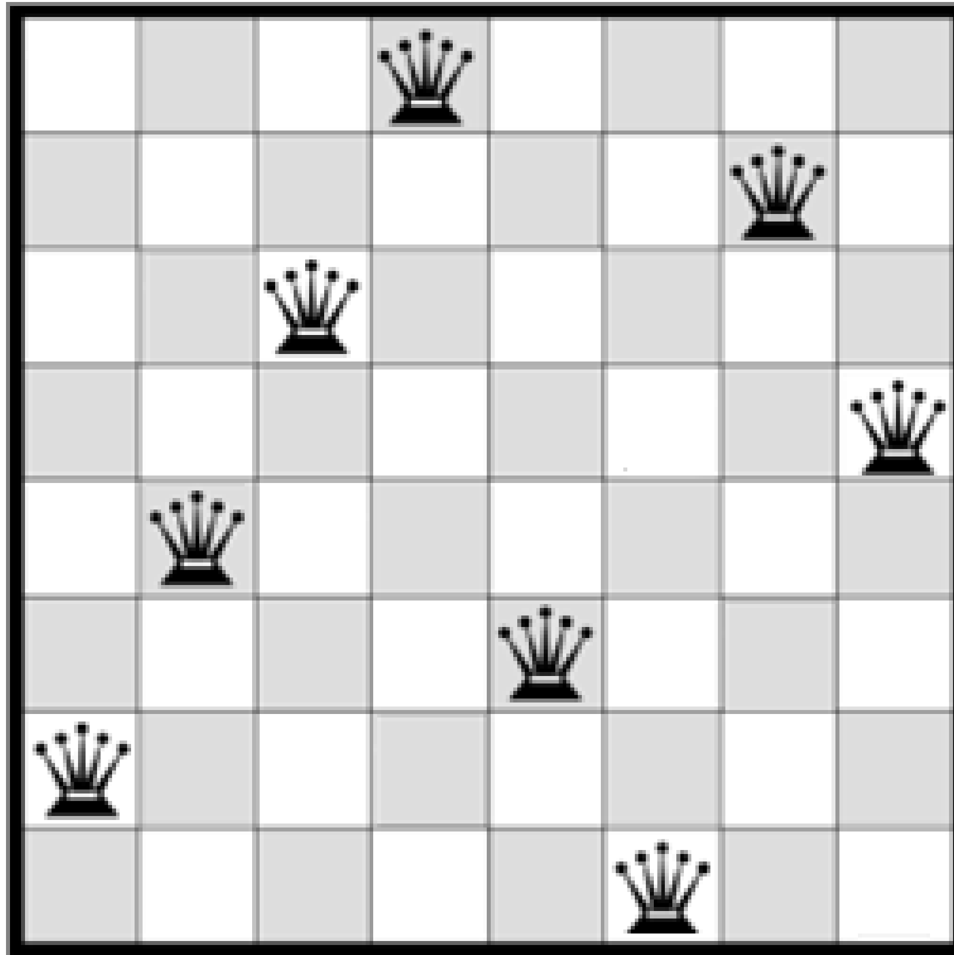
```
int main()
{
    int maze[HEIGHT][WIDTH] = {EMPTY};

    init_maze(maze);
    print_maze(maze);

    if (solve(maze, 0, 0, 1))
    {
        printf("Solution:\n");
        print_maze(maze);
    }
    else
        printf("No solution\n");

    return 0;
}
```

Eight Queens Puzzle

Place eight chess queens on an 8×8 chessboard so that no two queens attack each other.

A solution requires that no two queens share the same row, column, or diagonal.

Eight Queens Puzzle



Eight Queens Puzzle

Can you estimate how many distinct solutions exist?

There are **92** distinct solutions.

When uniting solutions that differ only by symmetry (rotations and reflections) – **12** solutions.

Searching for a solution - insights:

- There is exactly one queen in each row. Thus, in each step place a queen in the next row that does not contain a queen.
- In the next row - consider only the columns that are not already attacked by another queen.

Eight Queens Puzzle – Take 1

```
#define   SIZE      8
#define   EMPTY   0
#define   QUEEN  1

int  main()
{
    int board[SIZE][SIZE] = {EMPTY};

    if (solve(board))
            print_board(board);
    else
            printf("Failed to solve the %d queens problem\n");

return 0;
}
```

Eight Queens Puzzle – Take 1

```
int solve(int board[][SIZE])
{
    int row, col;

    for (row = 0; row < SIZE; ++row)
    {
        for (col = 0; col < SIZE; ++col)
        {
            if (check(board, row, col))
                board[row][col] = QUEEN;
        }
        if (col == SIZE)
            return FALSE;
    }
    return TRUE;
}
```

Eight Queens Puzzle – Take 1

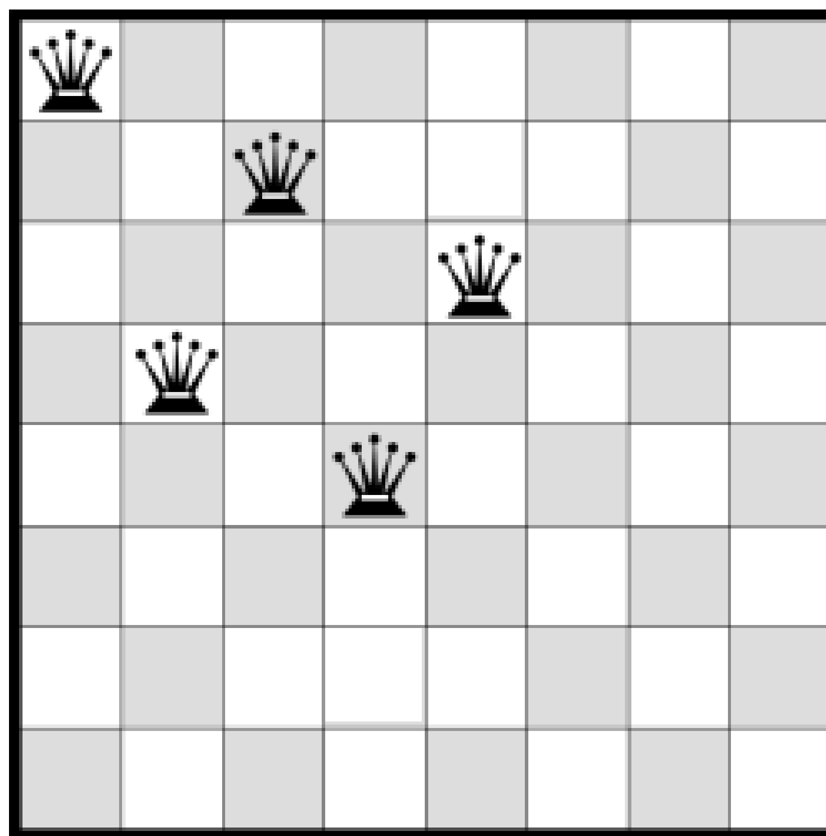
```
// helper - returns FALSE if a QUEEN is found on a path
// Prevents code duplication in check
int helper(int board[][SIZE], int row, int col, int add_row, int add_col)
{
    int i, j;

    i = row + add_row;
    j = col + add_col;
    while ((i >= 0) && (i < SIZE) && (j >= 0) && (j < SIZE))
    {
        if (board[i][j] == QUEEN)
                return FALSE;
        i += add_row;
        j += add_col;
    }
    return TRUE;
}
```

Eight Queens Puzzle – Take 1

```
int check(int board[][SIZE], int row, int col)
{
    return
    // check column
    (helper(board, row, col,  1, 0) && helper(board, row, col, -1,  0) &&
    // check row
    helper(board, row, col,  0, 1) && helper(board, row, col,  0, -1)  &&
    // check diagonal
    helper(board, row, col,  1, 1) && helper(board, row, col,  1, -1)  &&
    // check other diagonal
    helper(board, row, col, -1, 1) &&helper(board, row, col, -1, -1));
}
```

Eight Queens Puzzle – Problem

After adding 5 queens we get:



Eight Queens Puzzle – Solution

Add Recursion!

When there is no solution, return to the last added Queen and move her to the next possible cell.

If no such cell exists – move one Queen up.

- Change *solve* to be recursive. Now *solve* receives the board and the row number.

Eight Queens Puzzle – Solution

```
int solve(int board[][SIZE], int row)
{
    int col;

    if (row == SIZE)
        return TRUE;
    for (col = 0; col < SIZE; ++col)
        if (check(board, row, col))
        {
            board[row][col] = QUEEN;
            if (solve(board, row + 1)) return TRUE;
            // else - reset last assignment, try next possible column
            board[row][col] = EMPTY;
        }
    return FALSE;
}
```

Solution

```
#include <stdio.h>

#define SIZE 4

int main()
{
    int A[SIZE ][SIZE ] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12}, {13, 14,15, 16}};
    int i, j, B[SIZE ][SIZE ];
    // Compute the transpose
    for (i = 0; i < SIZE ; ++i)
            for (j = 0; j < SIZE ; ++j)
                    B[i][j] = A[j][i];
    // Print B
    for (i = 0; i < SIZE ; ++i)
    {
            for (j = 0; j < SIZE ; ++j)
                    printf("%3d ", B[i][j]);
            printf("\n");
    }
    return 0;
}
```