Survey of Scientific Computing

(SciComp 301)

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 22

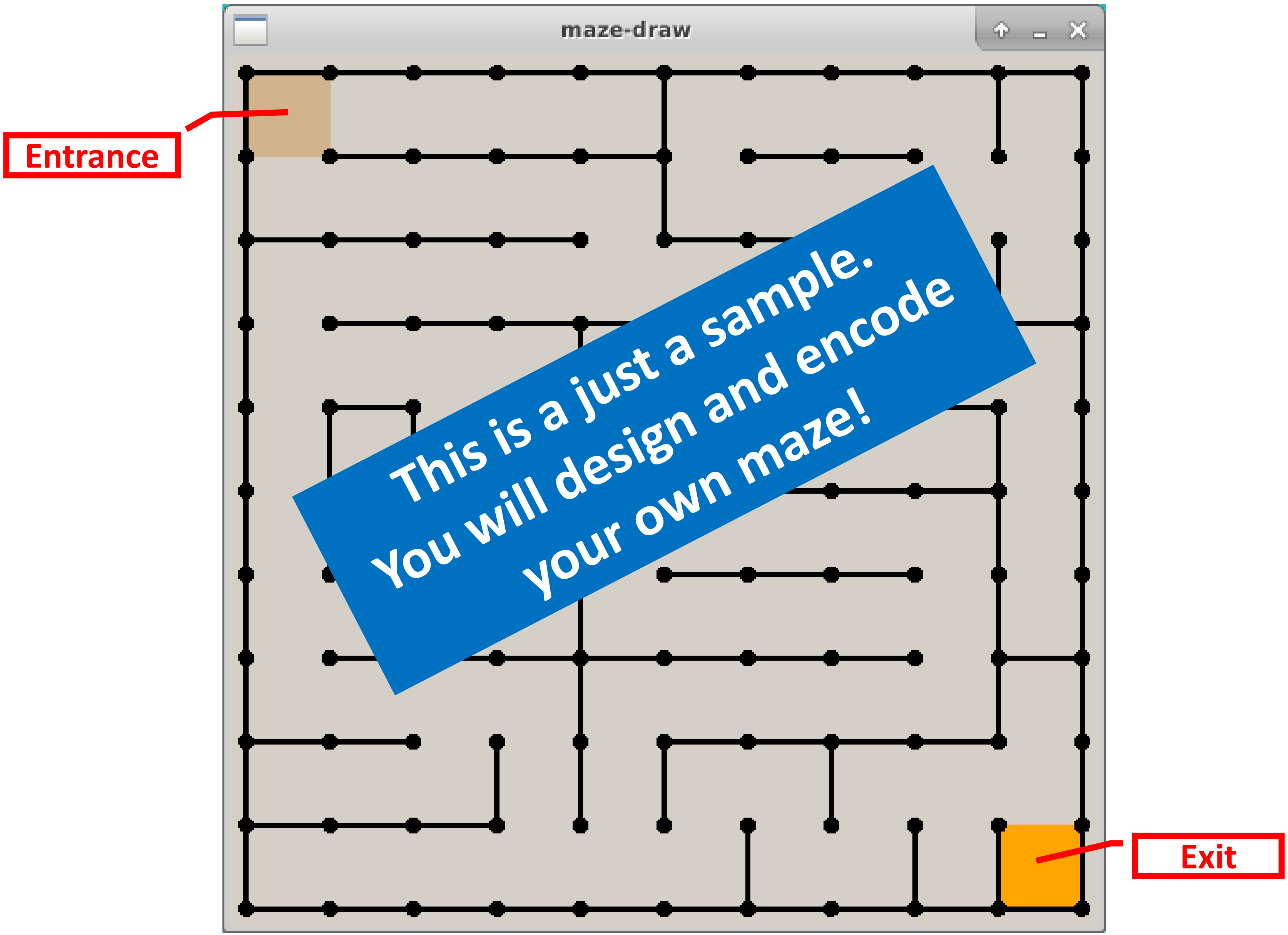
Search Algorithms, Adjacency Matrix

Session Goals

- Create a **2D maze** on graph paper
- Understand how to **encode** the cell walls in **base 2**
- Learn how **bitwise operators** can decode a wall value
- Perform **file input / output** using CSV and binary formats
- Appreciate backtracking in **depth-first** search algorithm
- Implement breadcrumbs using a **stack** data structure
- Learn how to multiply two matrices
- Create an **adjacency matrix** to improve search efficiency

Maze Solver

- Write a program that will find a path from the entrance to the exit of a given maze
  - The maze should be created using a simple **encoding**
  - During the search, the program cannot receive any hints from humans – it must run <u>autonomously</u>
- The maze is 10x10 cells, and **must have** at least <u>one</u> open path from entrance to exit cell
- The complete maze perimeter **must not** have <u>any</u> holes – all outside edges must be walls


maze-draw
Entrance
Exit
This is a just a sample.
You will design and encode
your own maze!

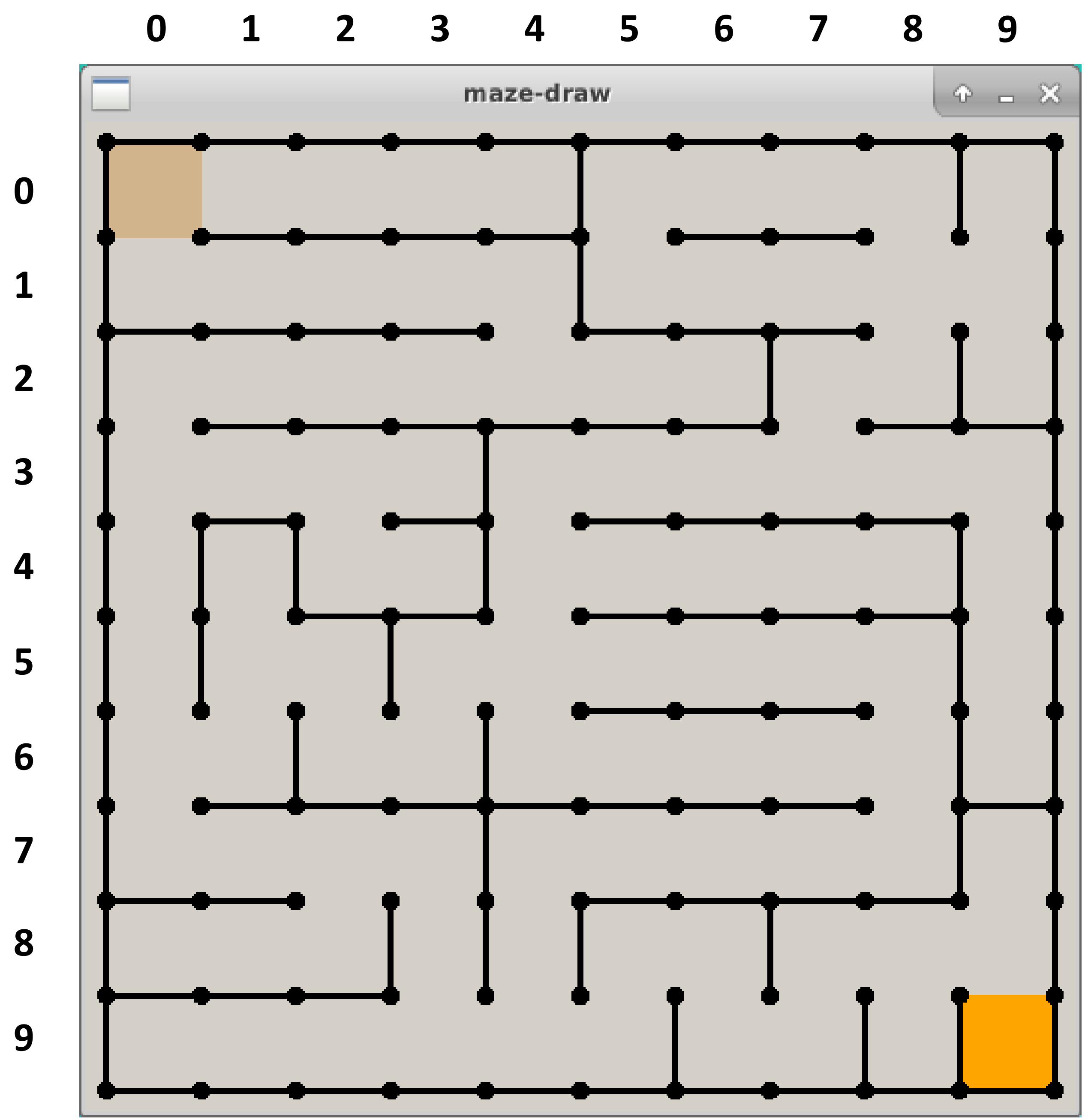
About Matrices

- Matrices are named <u>first</u> by **row**, *then* by **column**
  - A (**3 x 2**) matrix has **3** rows and **2** columns
- In **Cartesian** coordinates
  - The abscissa **X** is listed first, followed by the ordinate **Y**: **(X, Y)**
  - Abscissa++ moves the world point **right**
  - Ordinate++ moves the world point **up**
- In **Matrix** coordinates {an ***array*** in C++}
  - The **row** is listed first, followed by the **column**: **[Row] [Col]**
  - Row++ moves the cell **down** (inverted like screen coordinates!)
  - Col-- moves the cell to the previous element in the current Row

About Matrices

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	0,0	0,1	0,2	0,3	0,4
Row 1	1,0	1,1	1,2	1,3	1,4
Row 2	2,0	2,1	2,2	2,3	2,4

C++ arrays are 0-based



10 Rows
10 Cols

Y = **Rows**
X = **Cols**

Positive Y is down!

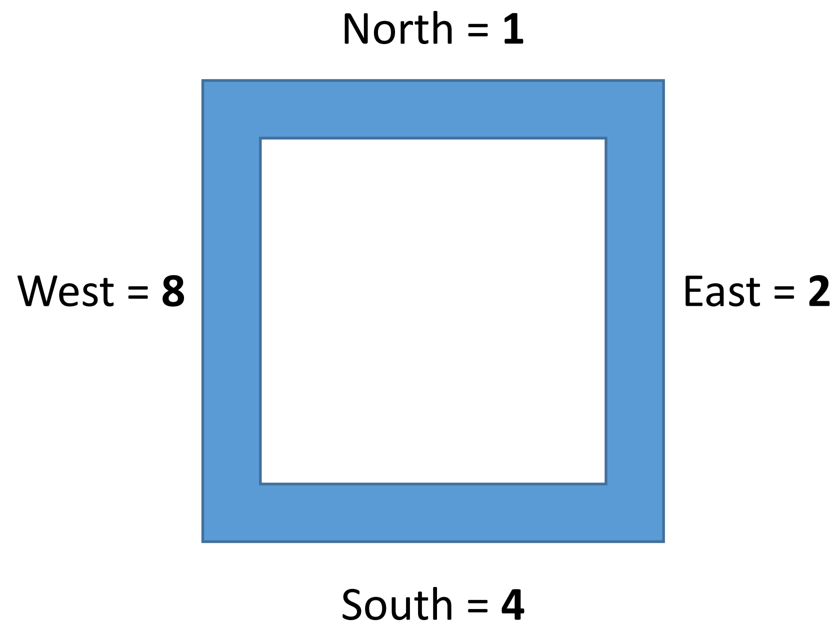
Entrance = [0][0]

Exit = [9][9]

How do we encode a maze?

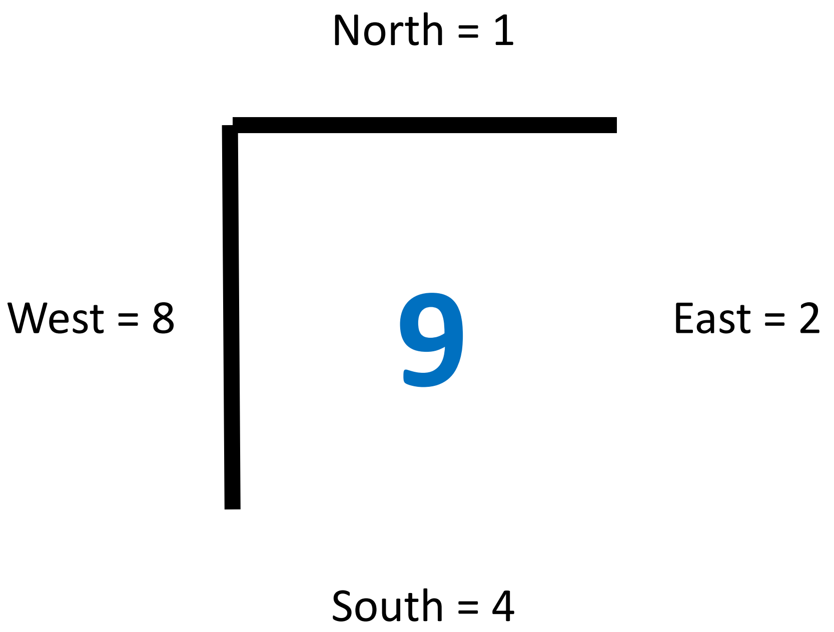
- Consider how to describe each individual square (cell) within the maze
- **How can we indicate for each individual cell if a wall exists to the North, East, South, or West direction?**

Encode each wall position as an increasing **power of 2**



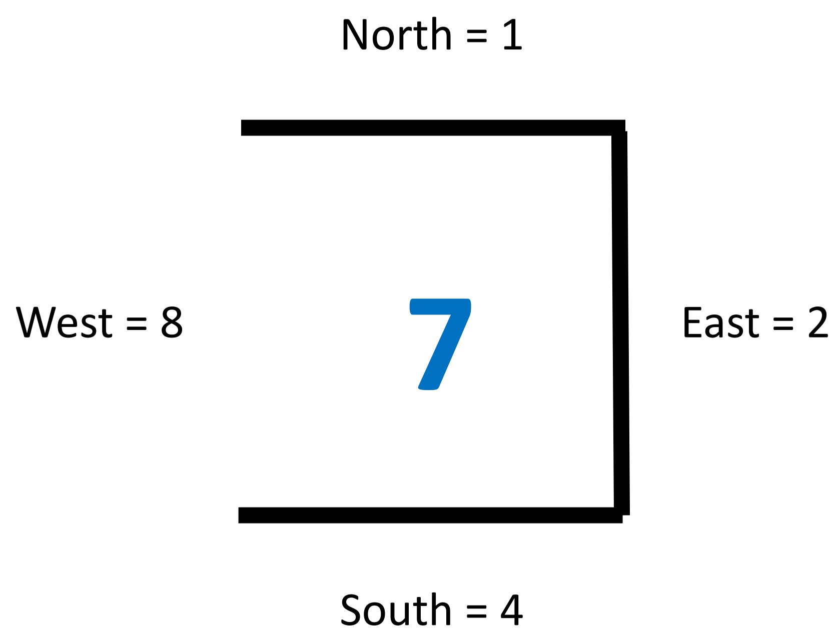
How do we encode a maze?

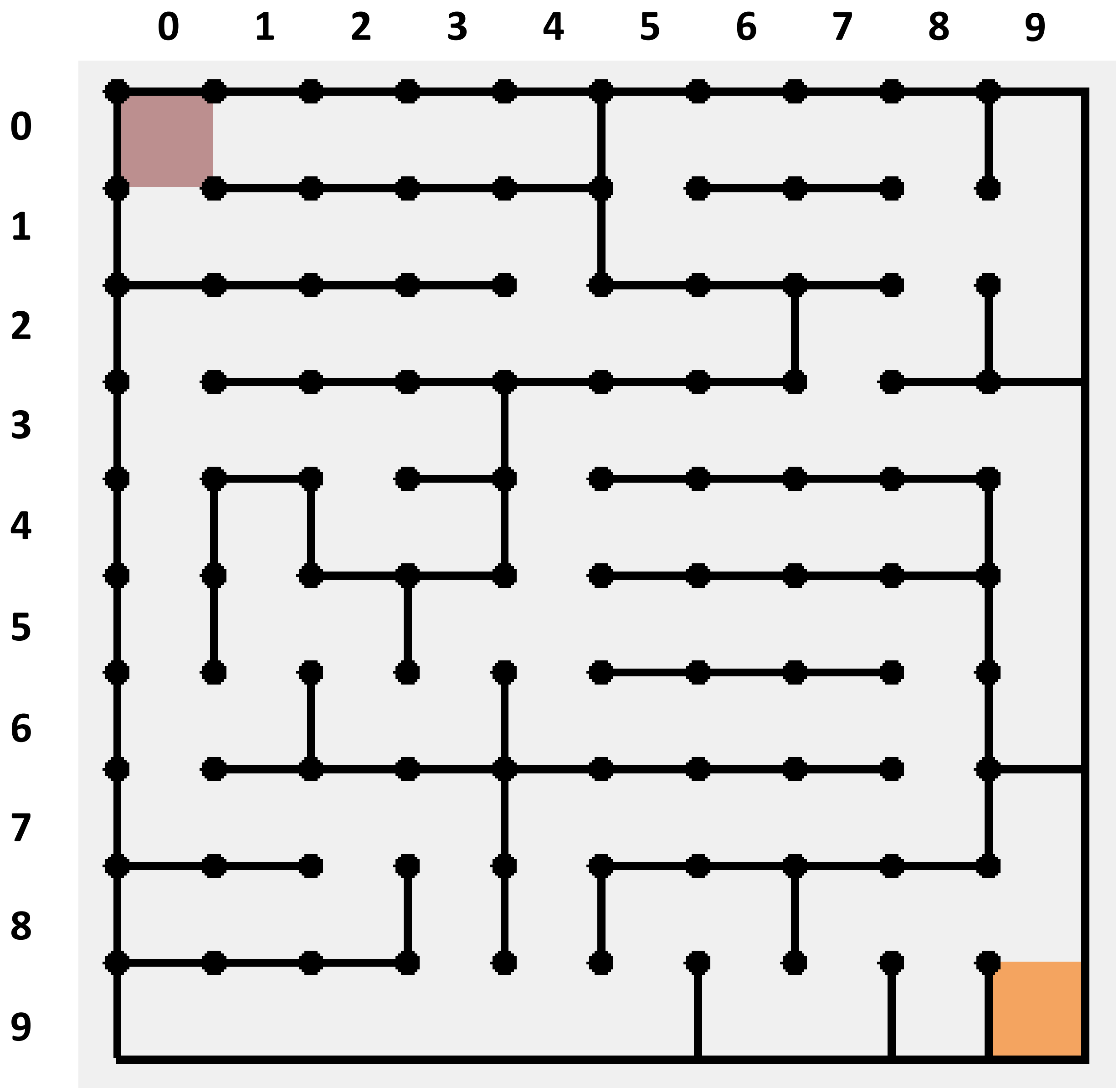
- If a wall exists in a given direction, add the value of that direction to the ***total*** cell value

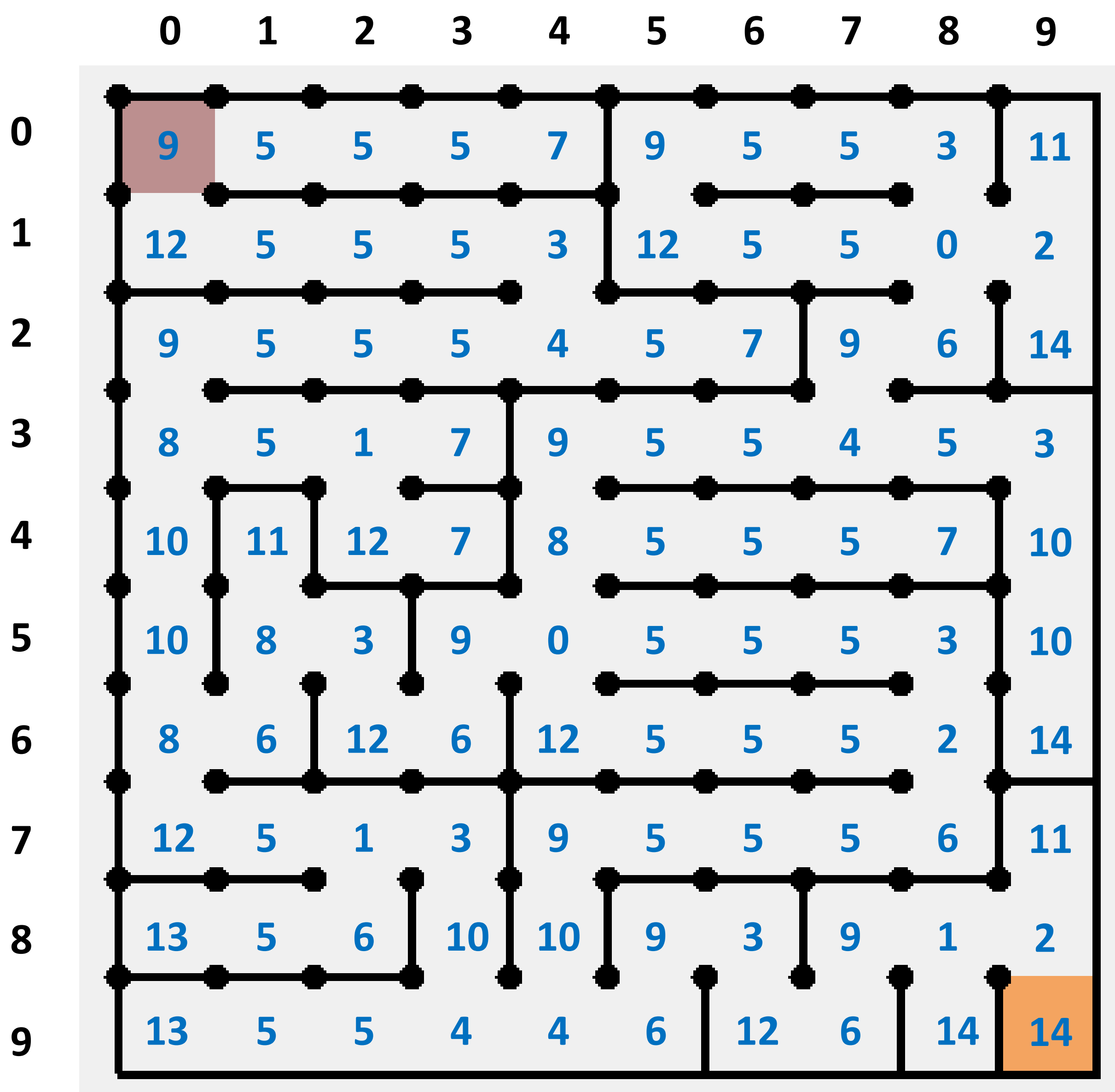


How do we encode a maze?

- Using a **power of 2** for each wall value produces an **unambiguous** encoding of all wall permutations
- 0 ≤ cell value ≤ 15
- No walls = zero (0)
- 15 = a totally "walled-in" cell that is unreachable ***(Hint: don't make a cell = 15)***




0 1 2 3 4 5 6 7 8 9
0
1
2
3
4
5
6
7
8
9


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

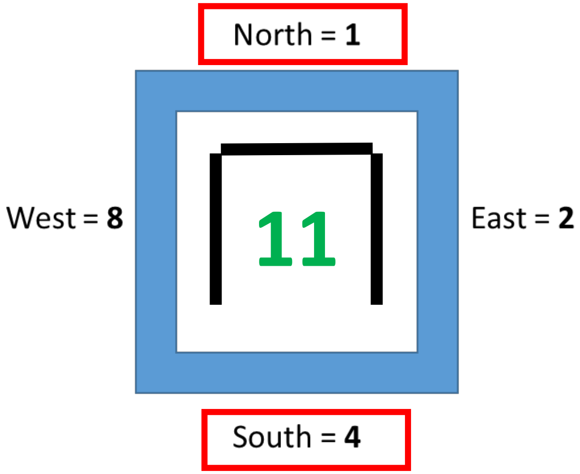
Drawing the 2D Maze

- Given a maze initialized with Base 2 wall encodings, how can we draw it?
- We could use a long series of **if()** statements to test all **16** possible values for a cell, and draw the required walls
- However, this would be <u>inefficient</u> in code size and run time
- We can take advantage of **bitwise AND** to figure out what walls to draw for a given cell

Bitwise Operators

- Logical operators use **double** ampersand or pipe characters
  - Boolean X **and** Y conditions: if (X && Y)
  - Boolean X **or** Y conditions: if (X || Y)
- ***Bitwise*** operators use **single** ampersand or pipe character
  - Bitwise X **and** Y values: X & Y
  - Bitwise X **or** Y values: X | Y
- Logical operators only return **true** or **false**
- **Bitwise** operators return a **integer value**
  - You normally only perform bitwise operations with **byte** or **int**

Bitwise AND



Binary ($Base_2$) Encoding

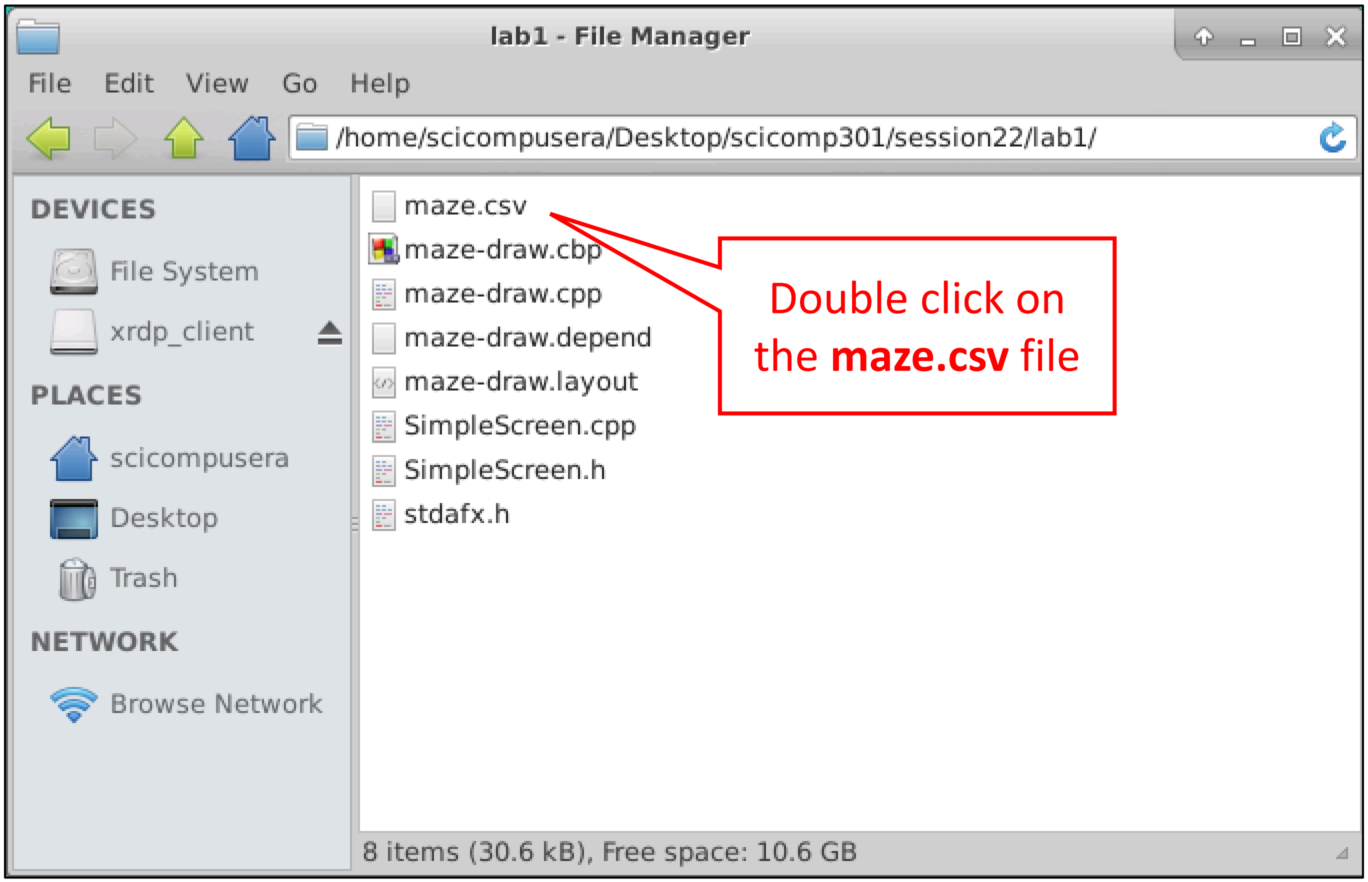
Position	3	2	1	0
Value	**8**	**4**	**2**	**1**

$Base_{10}$

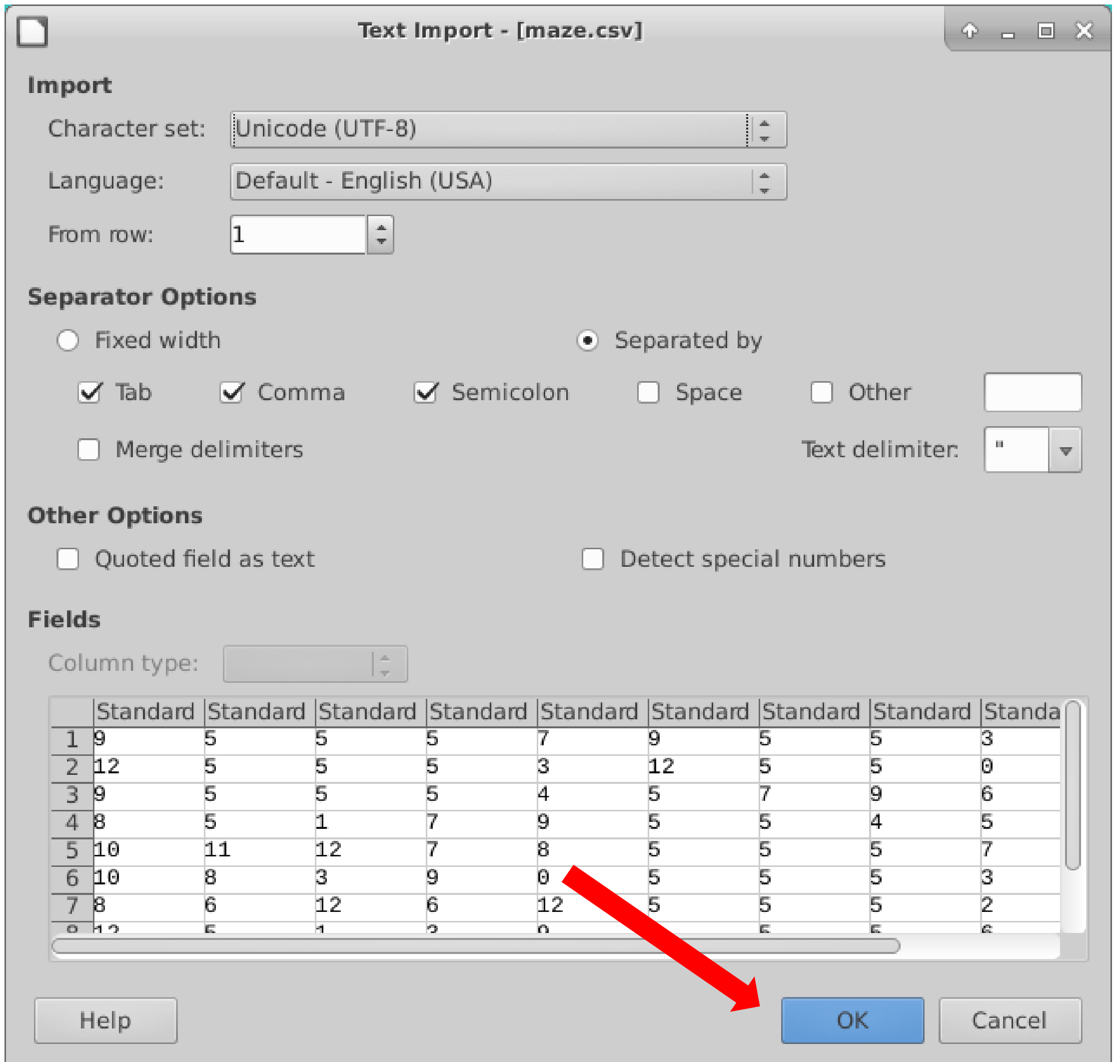
	1	0	1	1	11
AND	0	0	0	1	1
	0	0	0	1	**1**

	1	0	1	1	11
AND	0	1	0	0	4
	0	0	0	0	**0**

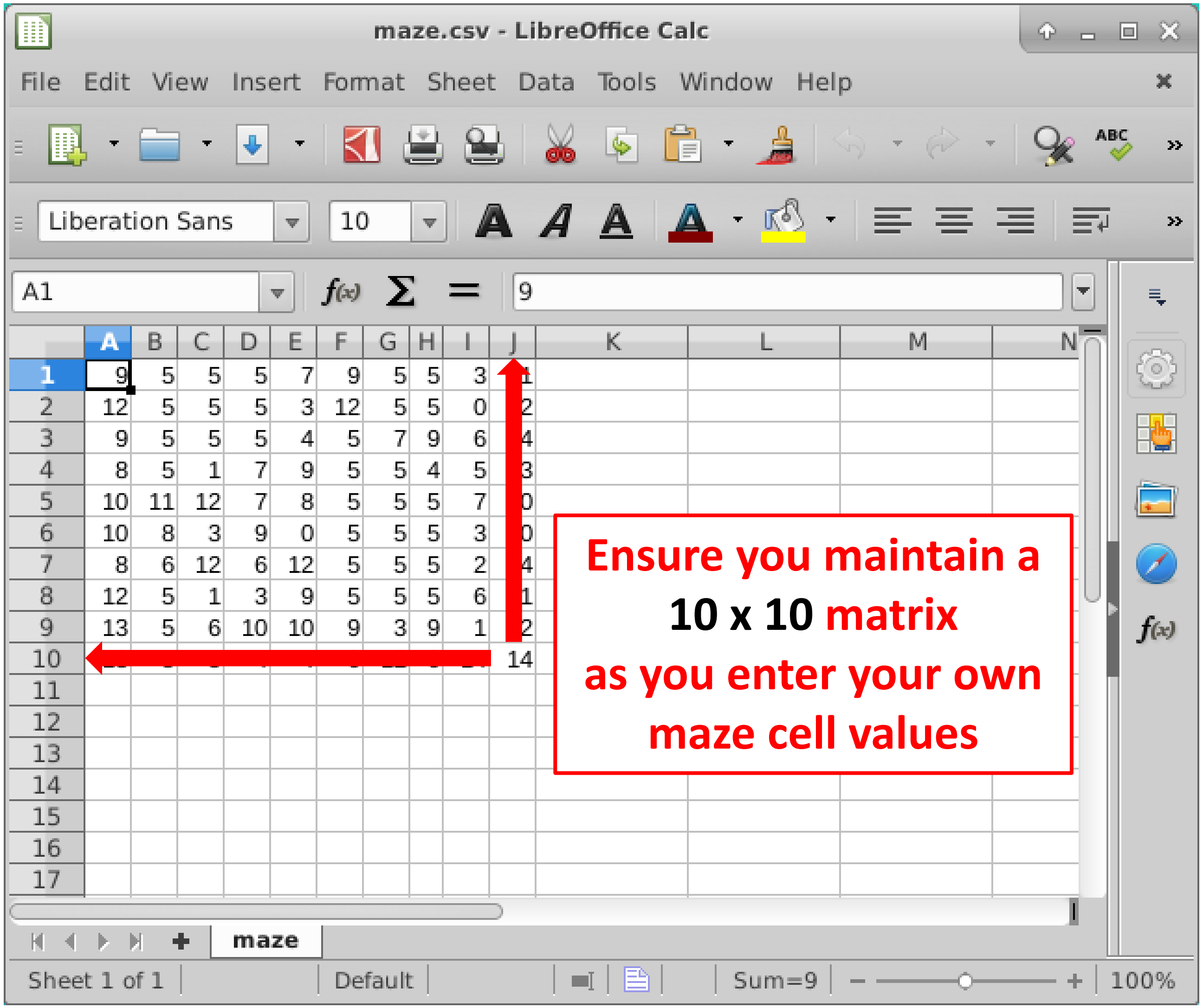
Encoding the Maze walls



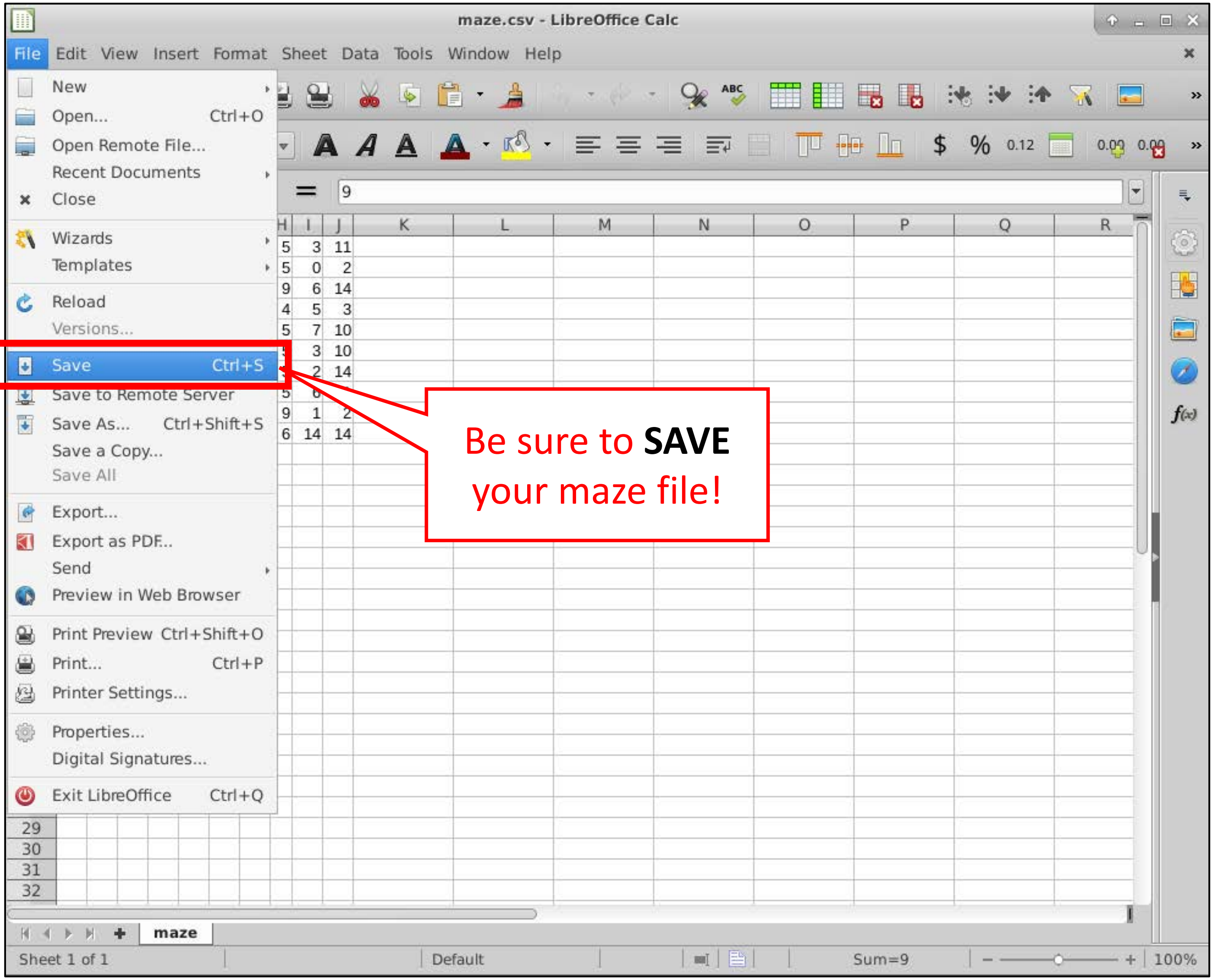
Encoding the Maze walls



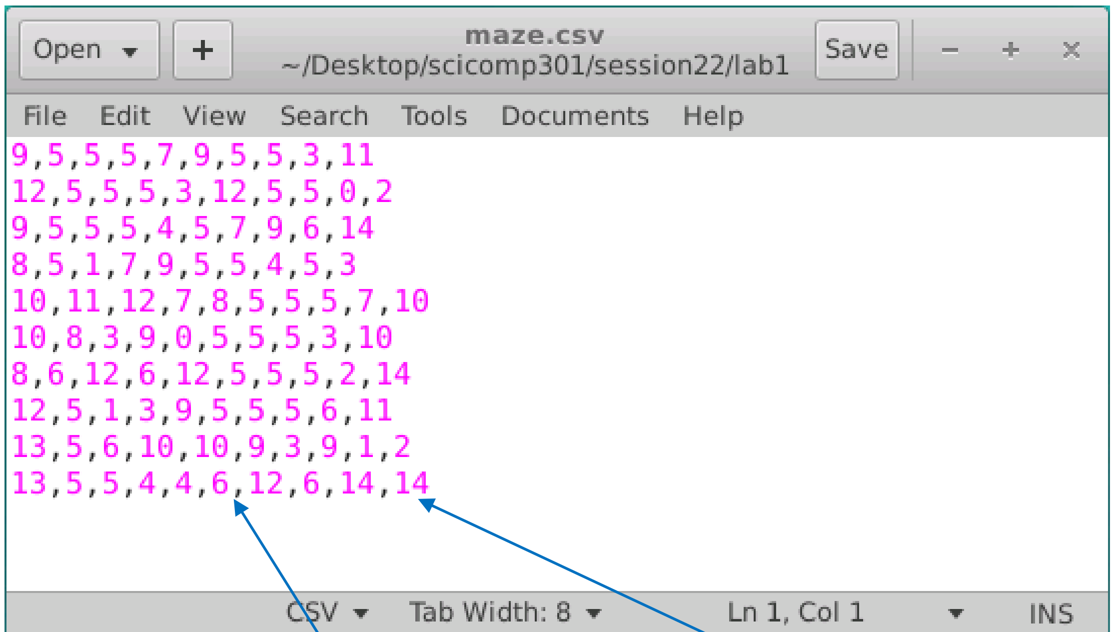
Encoding the Maze walls



Encoding the Maze walls



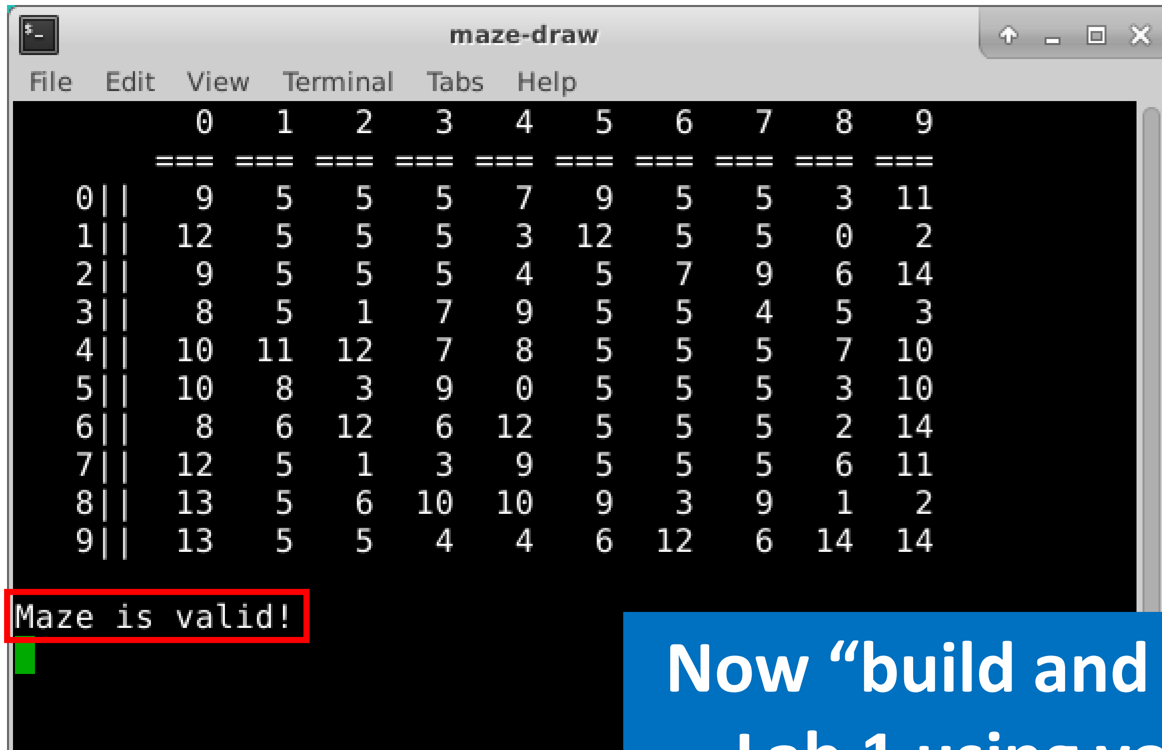
Encoding the Maze walls



CSV = Comma Separated Values

A "formatted" text file

Lab 1 – Maze Draw



Now "build and run" Lab 1 using your maze.csv file

Lab 1
Maze Draw

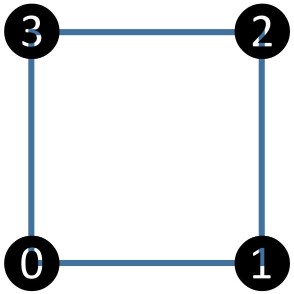
```
void draw(SimpleScreen& ss)
{
    // Draw maze (rows by cols)
    for (int r = 0; r < 10; r++) {
        double y0 = (9 - r) * 45;
        double y1 = (9 - r) * 45 + 45;
        for (int c = 0; c < 10; c++) {
            double x0 = c * 45;
            double x1 = c * 45 + 45;

            Point2D v0(x0, y0); // Lower-left vertex
            Point2D v1(x1, y0); // Lower-right vertex
            Point2D v2(x1, y1); // Upper-right vertex
            Point2D v3(x0, y1); // Upper-left vertex

            // Draw entrance cell
            if (r == 0 && c == 0)
                ss.DrawRectangle("tan", v0.x, v0.y, 45, 45, 1, true);
            // Draw exit cell
            if (r == 9 && c == 9)
                ss.DrawRectangle("orange", v0.x, v0.y, 45, 45, 1, true);

            // Draw cell corner circles
            ss.DrawCircle(v0.x, v0.y, 2, "black", 5);
            ss.DrawCircle(v1.x, v1.y, 2, "black", 5);
            ss.DrawCircle(v2.x, v2.y, 2, "black", 5);
            ss.DrawCircle(v3.x, v3.y, 2, "black", 5);

            int cell = maze[r][c];
            // Draw north wall if required
            if ((cell & 1) == 1)
                ss.DrawLine(v2, v3, "black", 3);
            // Draw west wall if required
            if ((cell & 2) == 2)
                ss.DrawLine(v1, v2, "black", 3);
            // Draw south wall if required
            if ((cell & 4) == 4)
                ss.DrawLine(v0, v1, "black", 3);
            // Draw east wall if required
            if ((cell & 8) == 8)
                ss.DrawLine(v0, v3, "black", 3);
        }
    }
}
```



Reading CSV files

```
struct csv_reader : ctype<char> {
    csv_reader() : ctype<char>(get_table()) {}
    static ctype_base::mask const* get_table() {
        static vector<ctype_base::mask>
            rc(table_size, ctype_base::mask());
        rc[','] = ctype_base::space;
        rc['\n'] = ctype_base::space;
        rc[' '] = ctype_base::space;
        return &rc[0];
    }
};
```

```
int maze[10][10];

void LoadMaze()
{
    string line;
    stringstream ss;
    ss.imbue(locale(locale(), new csv_reader()));

    ifstream mazeFile("maze.csv");
    if (!mazeFile) {
        cout << "Missing maze.csv file!" << endl;
        exit(-1);
    }

    for (int r = 0; r < 10; r++) {
        getline(mazeFile, line);
        ss.str(line);
        for (int c = 0; c < 10; c++)
            ss >> maze[r][c];
        ss.clear();
    }

    mazeFile.close();
}
```

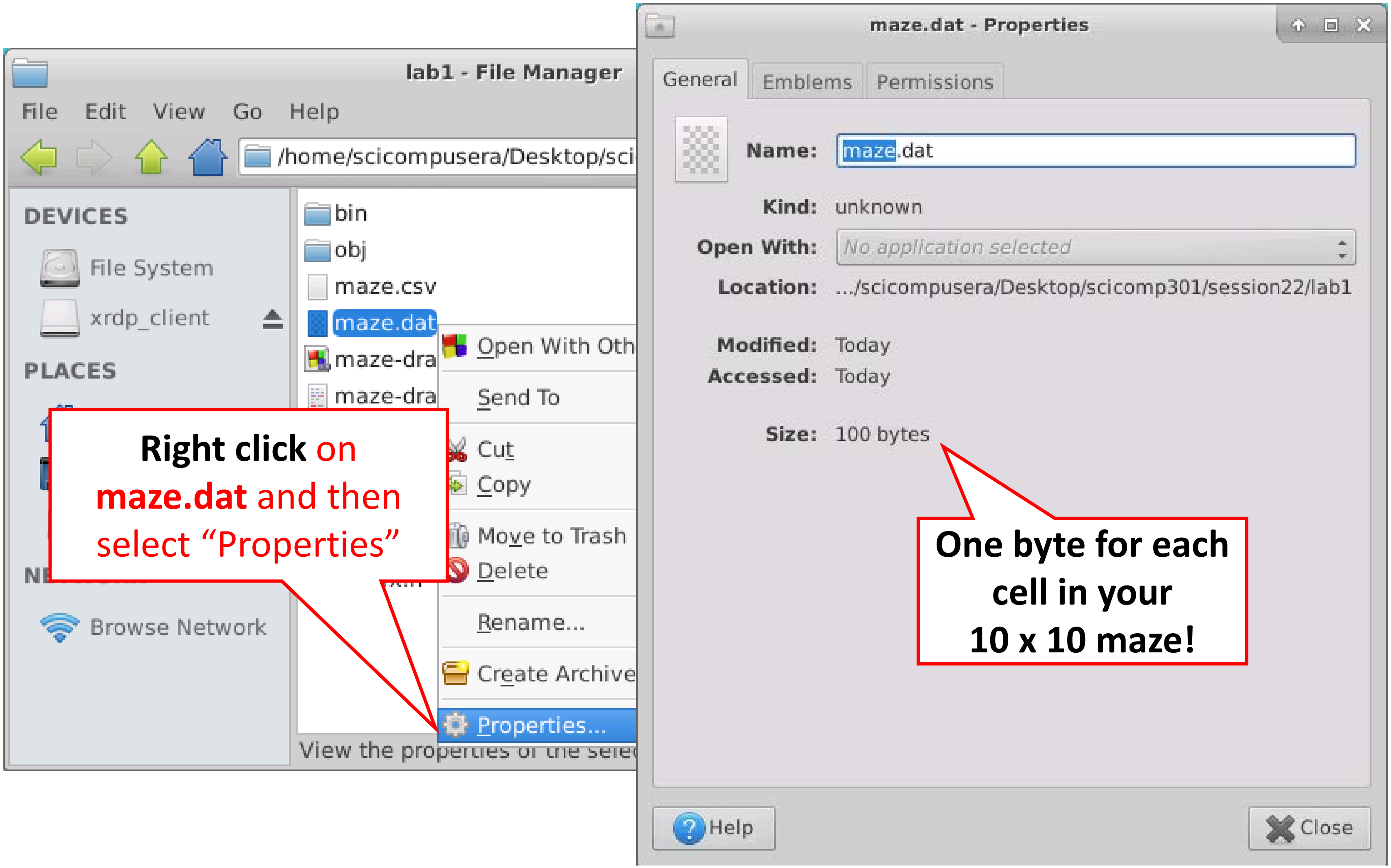
Writing data in BINARY format

```
void SaveMaze()
{
    ofstream mazeFile("maze.dat", ios::binary);

    for (int r = 0; r < 10; r++)
        for (int c = 0; c < 10; c++)
            mazeFile.write((char*)&maze[r][c], sizeof(char));

    mazeFile.close();
}
```

Lab 1 – Binary Output File



Lab 1 – Binary Output File

You must copy your maze.dat file to the lab2 and lab3 folders!

Depth-First Search

- Depth-first is a sequential search algorithm
  - It is just you alone in the maze, you have no helpers
  - It is a zero *prior* knowledge, recursive, backtracking approach
  - You have **breadcrumbs** to mark your cell visitation history
- Order of step search in each cell is **North, East, South, West**
  - We can only proceed in a direction if there is no wall in the path:

if (cell value & direction) != direction

North (1)	→	(Δ row = -1, Δ column = 0)
East (2)	→	(Δ row = 0, Δ column = 1)
South (4)	→	(Δ row = 1, Δ column = 0)
West (8)	→	(Δ row = 0, Δ column = -1)

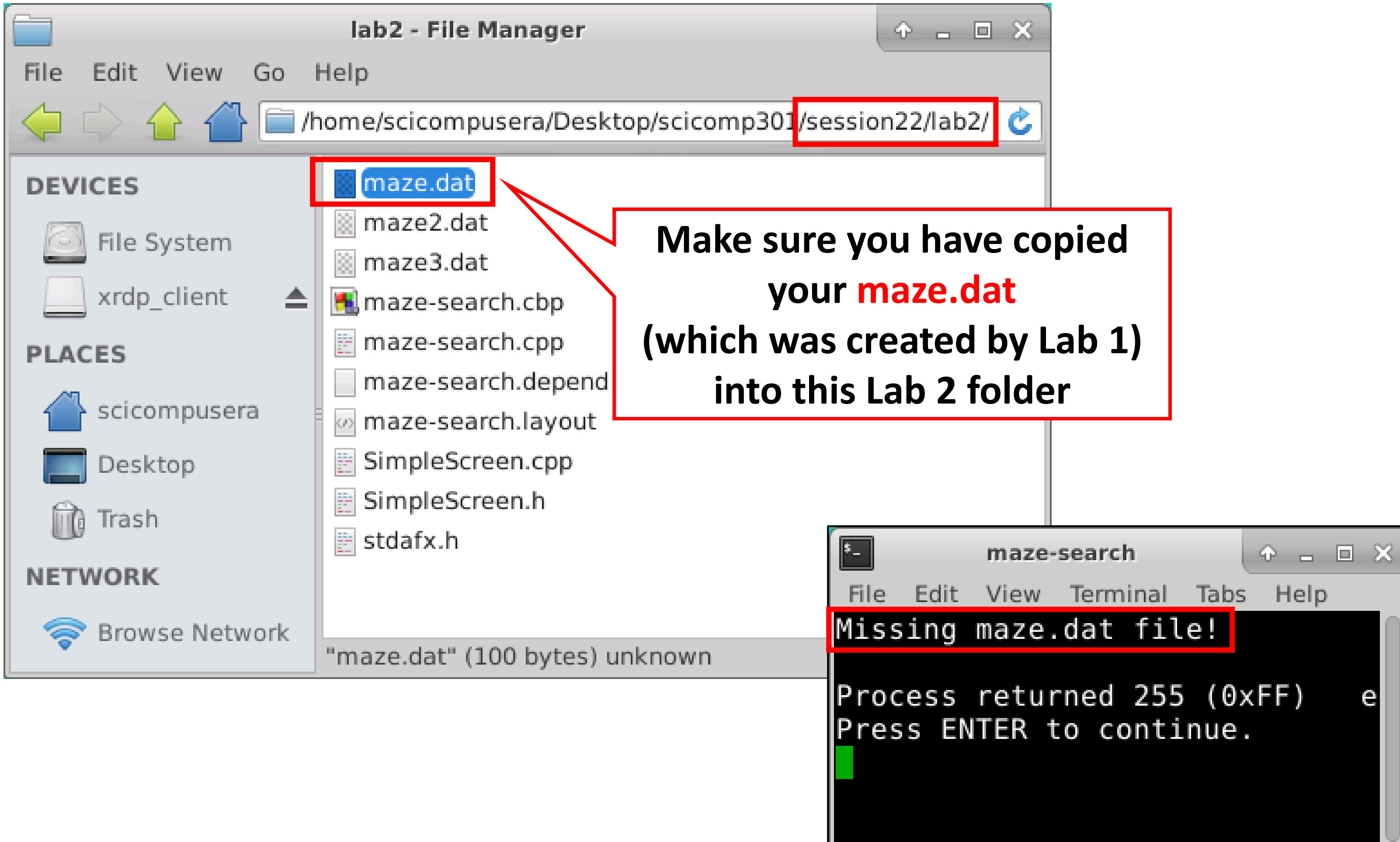
Depth-First Search Algorithm

1. Drop a **breadcrumb** as you enter each cell
2. Take a step in the very first direction that is **open**, and go to step #1
3. If there are no more open directions in the cell, retrace your steps **backwards** until you reach a cell with a breadcrumb where **the next open direction** is one you have not taken yet
4. Take a step in that new direction, and go to step #1
5. Stop with you reach the **exit square**

Depth-First Search Breadcrumbs

- A **breadcrumb** matrix (array) can contain a simple bool value to indicate if you have previously visited this cell
- Breadcrumbs prevent going around in endless circles and never finding the exit
- In this program we use an **bool** **visitCount** array, so we can color the path according to the number of times we've visited each cell (1=**Blue**, 2=**Green**, 3=**Red**, 4=**Orange**)
- Lots of **red** and **orange** squares in the path indicates an *inefficient* search pattern, because you are visiting the same node too many times!

Lab 2 – Maze Search



Lab 2 – Maze Search

```
void LoadMaze()
{
    ifstream mazeFile("maze.dat", ios::binary);
    if (!mazeFile) {
        cout << "Missing maze.dat file!" << endl;
        exit(-1);
    }

    for (int r = 0; r < 10; r++)
        for (int c = 0; c < 10; c++)
            mazeFile.read((char*)&maze[r][c], sizeof(char));

    mazeFile.close();
}
```

Lab2 reads in the binary file maze.dat

```cpp
bool TakeStep()
{
    int r = get<0>(stack.back());
    int c = get<1>(stack.back());
    int r2 = r; int c2 = c;
    int dir = get<2>(stack.back());
    if (dir == 1) r2--;
    if (dir == 2) c2++;
    if (dir == 4) r2++;
    if (dir == 8) c2--;
    get<2>(stack.back()) *= 2;
    bool moved = false;
    if (((maze[r][c] & dir) != dir) &&
        visitCount[r2][c2] == 0)
    {
        tuple<int, int, int> cell(r2, c2, 1);
        stack.push_back(cell);
        moved = true;
    }
    if (dir == 16 && !moved) {
        stack.pop_back();
        r2 = get<0>(stack.back());
        c2 = get<1>(stack.back());
        moved = true;
    }
    if (moved) {
        visitCount[r2][c2]++;
        totalSteps++;
        if (r2 == 9 && c2 == 9)
            foundExit = true;
        return true;
    }
    return false;
}
```

Lab 2
Maze Search

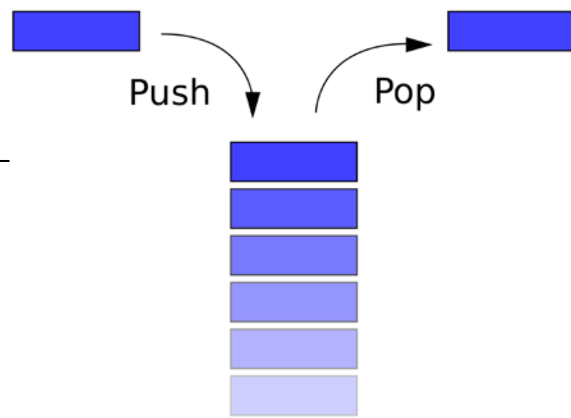
The program tries to take a step in each direction, updating the **visitCount** array with each step

Lab 2 – Maze Draw

Entrance

stack	Row	Col	Row	Col	Row	Col	Row	Col
Level 0	0	0	0	0	0	0	0	0
Level 1	0	1	0	1	1	0	1	0
Level 2	0	2	0	2	1	1	1	1
Level 3	0	3	0	3	1	2	1	2
Level 4	0	4			1	3	1	3
Level 5					1	4	1	4
Level 6					2	4	2	4
Level 7					2	5	2	5
Level 8					2	6		

Dead-end → pop_back() → Dead-end → pop_back()



We use a "stack" data structure to record the [row][col] of the navigator at each step through the maze

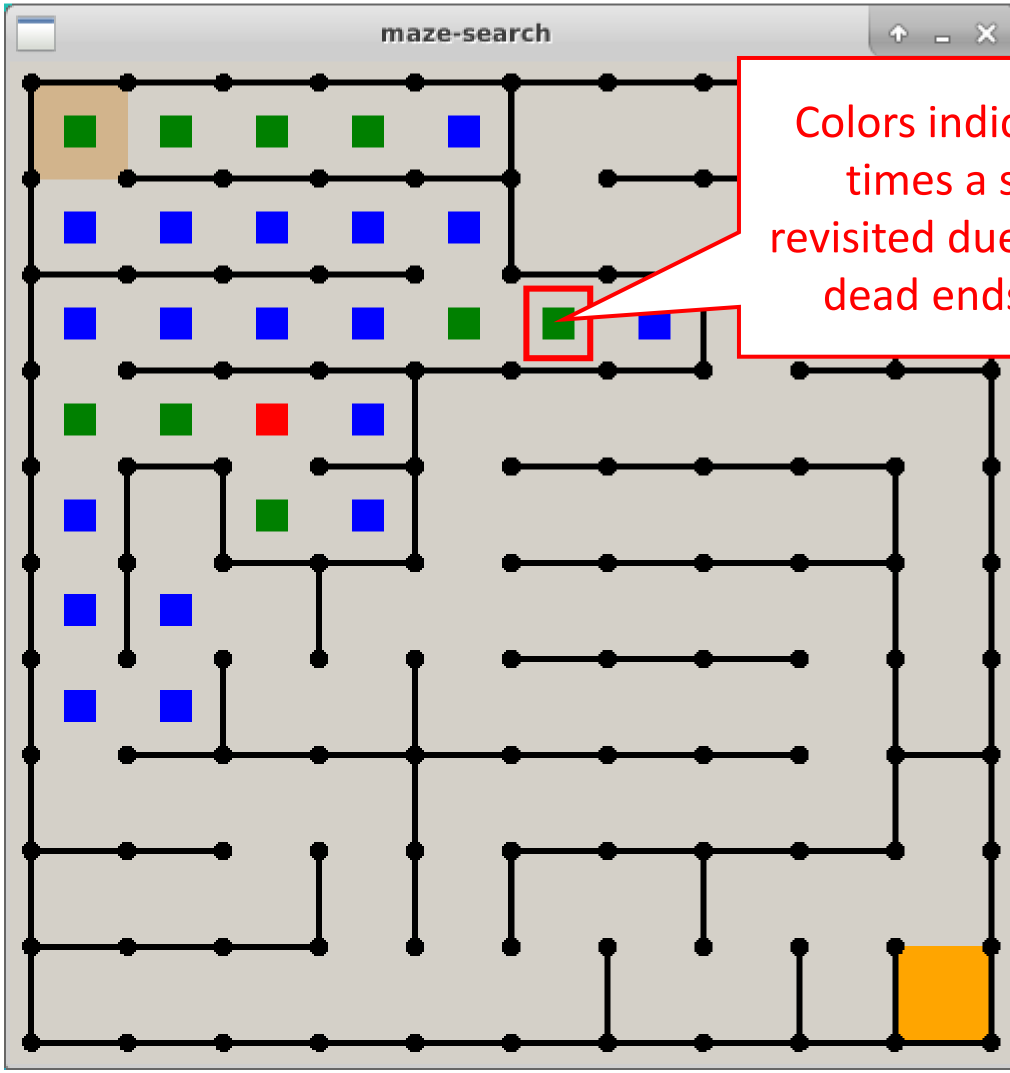
Lab 2 – Maze Search

```
void eventHandler(SimpleScreen& ss, ALLEGRO_EVENT& ev)
{
    if (ev.type == ALLEGRO_EVENT_KEY_CHAR) {
        if (ev.keyboard.keycode == ALLEGRO_KEY_S) {
            if (!foundExit) {
                while (!TakeStep());
                if (foundExit) {
                    cout << "Exit found!" << endl
                        << "Total steps = " << totalSteps << endl
                        << "Path steps = " << stack.size() - 1 << endl;
                    ResetVisitCount();
                    for (auto s : stack)
                        visitCount[get<0>(s)][get<1>(s)] = 1;
                    ss.Clear();
                    DrawMaze(ss);
                }
                ss.Redraw();
            }
        }
    }
}
```

Press S to take a single step

When the exit is found, your path will be shown (after removing any backup steps)

Lab 2 – Maze Search

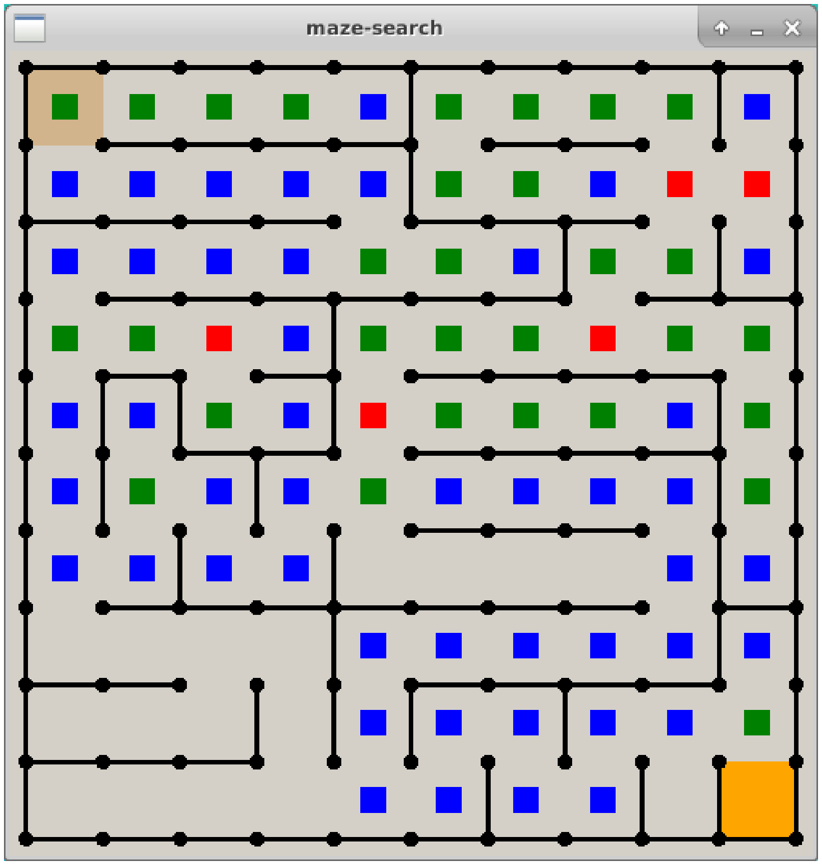


Colors indicate the number of times a square (cell) was revisited due to backing up from dead ends or breadcrumbs

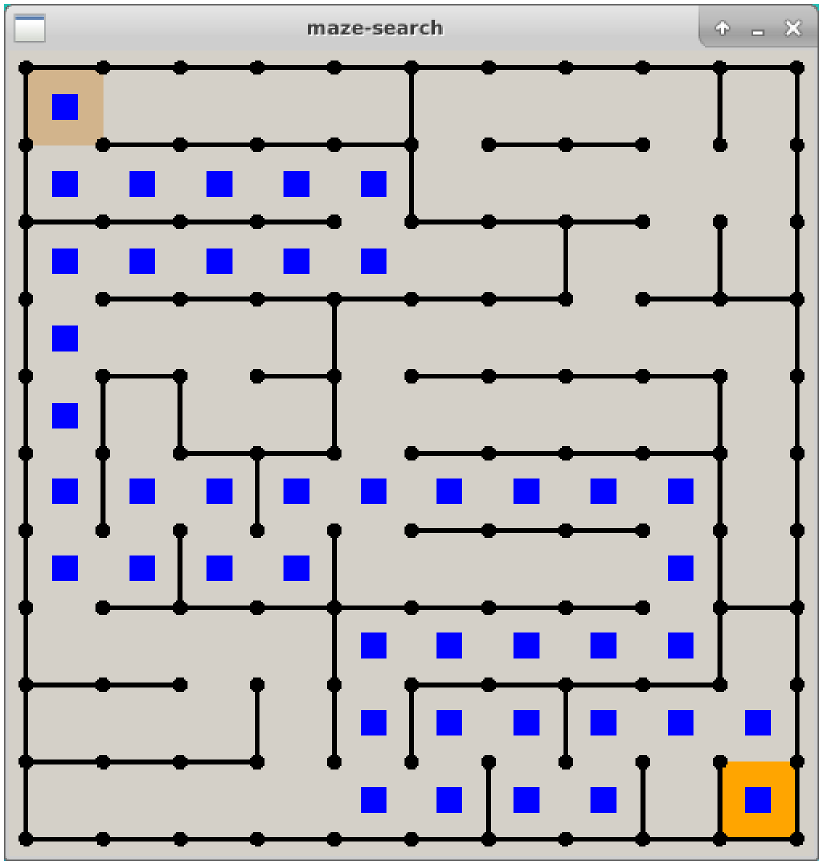
Press S to take a single step

Lab 2 – Maze Search

Right ***before*** stepping into exit cell

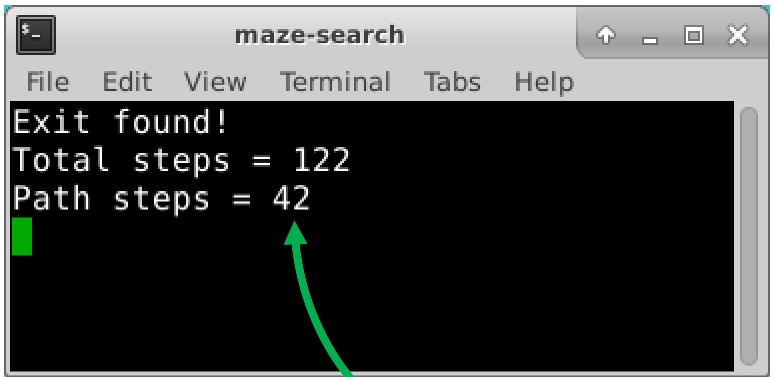


Right ***after*** stepping into exit cell



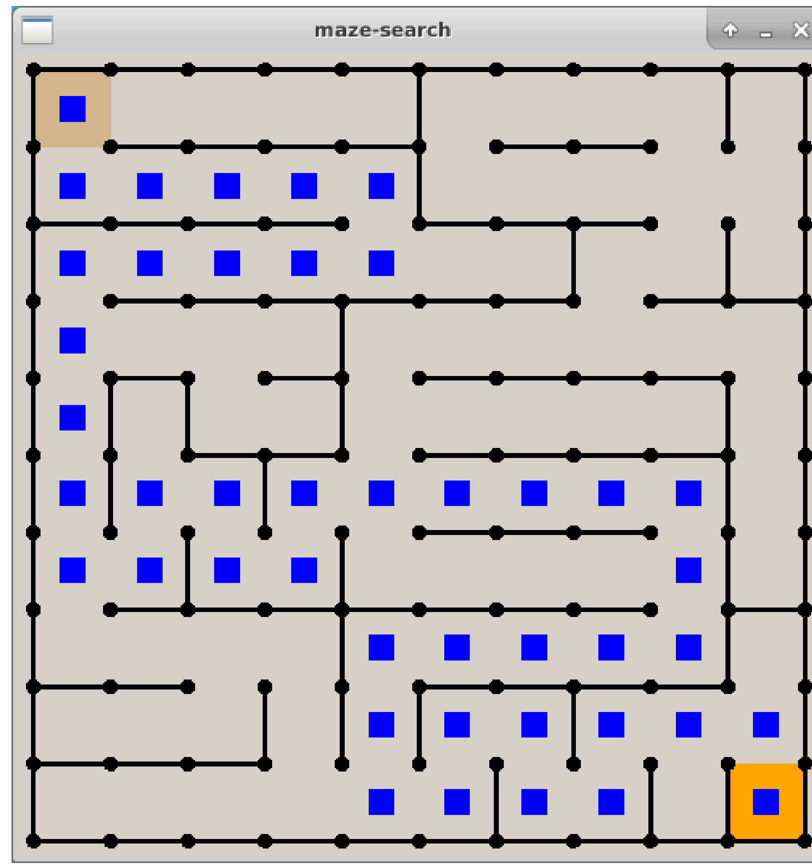
When the exit is found, your path will be shown (after removing any backup steps)

Lab 2 – Maze Search



- **Total steps** = How many you had to take counting backup steps
- **Path steps** = The **best path you found** minus any backup steps

Right ***after*** stepping into exit cell



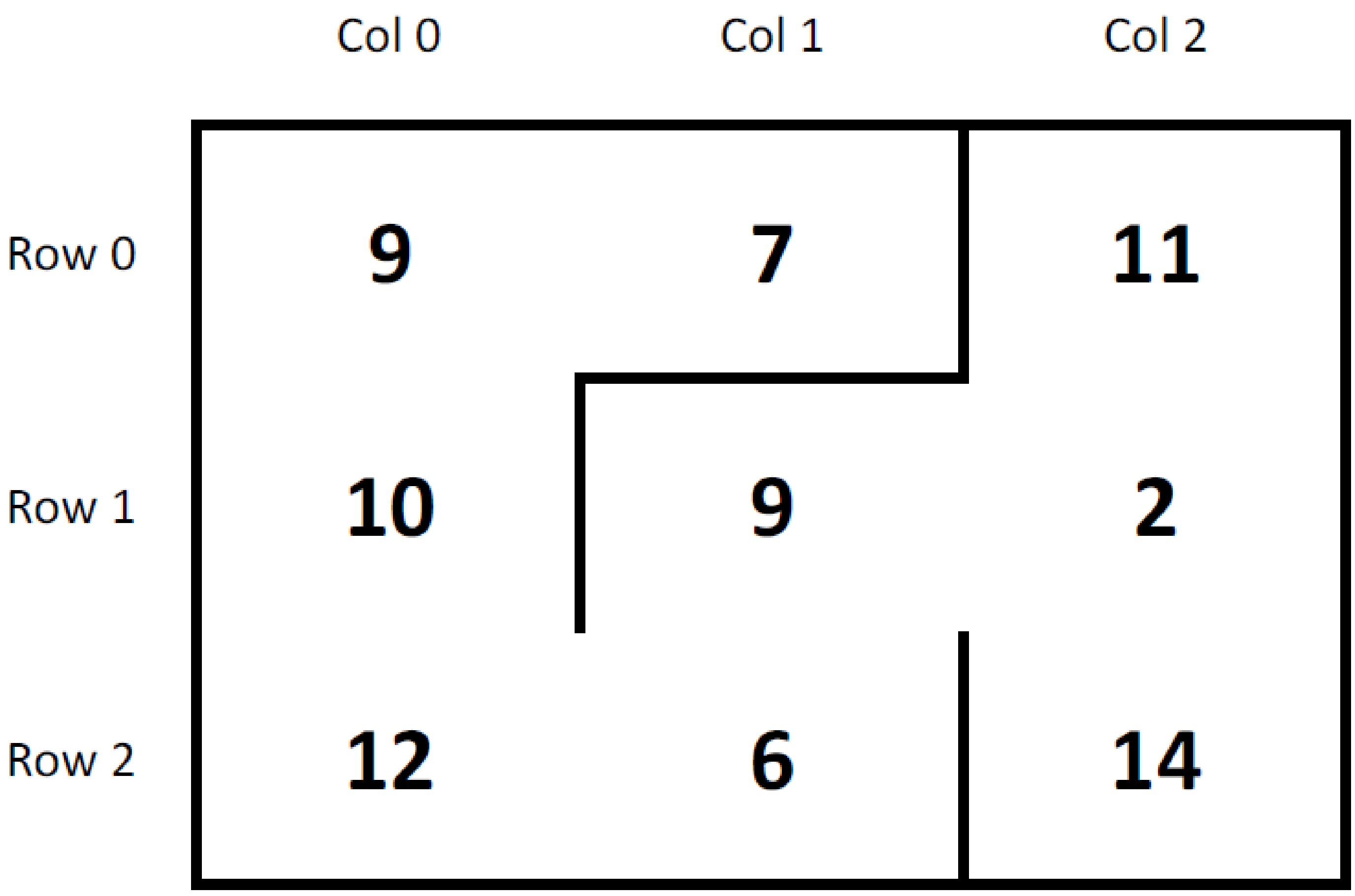
When the exit is found, your path will be shown (after removing any backup steps)

Improving Depth-First Search Efficiency

- The unaided depth-first search spends considerable time exploring paths which clearly are not on the *optimal* route – it hits a lot of **dead ends**
- What if we could calculate the **shortest path length**, and start **backtracking** the instant our current path length ≥ the *shortest* path length?
- It is possible to calculate the **minimum** number of steps from entrance to exit (***the*** shortest path) ***without searching one cell or taking one step!***

Adjacency Matrix

3 x 3 maze
9 total cells



Adjacency Matrix

Every maze square (cell) is represented along **both** the rows and columns of an adjacency matrix

Row	Col
0	0
0	1
0	2
1	0
1	1
1	2
2	0
2	1
2	2

Col	0	1	2	0	1	2	0	1	2
Row	0	0	0	1	1	1	2	2	2
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	1	1	0	1	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
2	0	0	1	0	0	1	0	0	0
3	1	0	0	1	0	0	1	0	0
4	0	0	0	0	1	1	0	1	0
5	0	0	1	0	1	1	0	0	1
6	0	0	0	1	0	0	1	1	0
7	0	0	0	0	1	0	1	1	0
8	0	0	0	0	0	1	0	0	1

A **true** (1) indicates you **can** reach the other cell in **just one step**
A **false** (0) means that you **cannot** reach the other cell in just one step

Adjacency Matrix

The main diagonal has all one values
because every cell can reach itself by definition

Row	Col
0	0
0	1
0	2
1	0
1	1
1	2
2	0
2	1
2	2

Col	0	1	2	0	1	2	0	1	2
Row	0	0	0	1	1	1	2	2	2
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	1	1	0	1	0	0	0	0	0
1	1	1	0	[illegible]	0	0	0	0	0
2	0	0	1	[illegible]	0	1	0	0	0
3	1	0	0	1	0	0	1	0	0
4	0	0	0	0	1	1	0	1	0
5	0	0	1	0	1	1	0	0	1
6	0	0	0	1	0	0	1	1	0
7	0	0	0	0	1	0	1	1	0
8	0	0	0	0	0	1	0	0	1

The whole matrix is symmetric about the main diagonal
because there are no “one-way” doors in the maze.
Reflexive Property: If you can get from cell A to cell B in one step, then you can also get from cell B to cell A in one step

Adjacency Matrix

- To find the **shortest path**, we keep **AND**ing (using the ***logical*** operator **&&**) the adjacency matrix against itself until a ***true*** value appears in the matrix element that represents the **exit cell**
- The number of times we had to "multiply" (AND) the adjacency matrix by itself **equals the minimal # of steps** from entrance to exit
- **Ironically, we can know the *# of steps* in the shortest path, but not what the actual steps are!**
- But how do we multiple two matrices together?

Matrix Multiplication

- In order to multiply 2-D matrices together the number of *columns* in matrix **A** must equal the number of *rows* in matrix **B** (Cols **A** = Rows **B**)
- The resulting matrix will have as many rows as there were <u>rows</u> in matrix **A**, and as many <u>columns</u> as there were in matrix **B** (Rows **A** x Cols **B**)

Matrix Multiplication

- Example
  - Matrix A has dimension (3 x 2) = 3 rows, 2 columns
  - Matrix B has dimension (2 x 3) = 2 rows, 3 columns

These values must match!

(3 x 2) • (2 x 3)

These two values will be the dimension of the matrix product!

Matrix Multiplication

- Matrix multiplication is **not** commutative!
  - If we multiple **B** x **A** we will get a different matrix than if we multiple **A** x **B**
  - Very strange – catches even great physicists by surprise!

    (3 x 2) • (2 x 3) = result is a (3 x 3) matrix

    (2 x 3) • (3 x 2) = result is a (2 x 2) matrix
- Welcome to the world of **non-commutative algebra**!
  - This asymmetry is the foundation of the matrix formulation of **quantum mechanics**

Matrix Multiplication

- The algorithm is simple but tedious for **A** x **B** = **C**
- **Sum** the **product** of every element in each row of matrix **A** multiplied by the corresponding element in each column of matrix **B**
- That *sum* becomes one element in new matrix **C**
- Continue this process for all rows in matrix **A**
  - Every row in **A** gets multiplied by every column in **B**
  - Output matrix has dimensions (Rows A **x** Cols B)

Matrix Multiplication

Matrix A
(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B
(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**
Product Cell (**2,1**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**
Product Cell (**1,2**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**
Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 9 x 1 + 7 x 9) = **76**

Matrix C
(2 rows x 2 cols)

78	**93**
91	**76**

Matrix Multiplication

Matrix A

(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B

(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**

Product Cell (**2,1**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**

Product Cell (**1,2**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**

Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 5 x 1 + 7 x 9) = **76**

Matrix C

(2 rows x 2 cols)

78	**93**
91	**76**

Matrix Multiplication

Matrix A

(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B

(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**

Product Cell (**2,1**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**

Product Cell (**1,2**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**

Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 9 x 1 + 7 x 9) = **76**

Matrix C

(2 rows x 2 cols)

78	**93**
91	**76**

Matrix Multiplication

Matrix A

(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B

(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**

Product Cell (**2,1**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**

Product Cell (**1,2**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**

Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 9 x 1 + 7 x 9) = **76**

Matrix C

(2 rows x 2 cols)

78	**93**
91	**76**

Matrix Multiplication

Matrix A

(2 rows x 3 cols)

	Col 1	Col 2	Col 3
Row 1	**4**	**5**	**8**
Row 2	**1**	**9**	**7**

X

Matrix B

(3 rows x 2 cols)

	Col 1	Col 2
Row 1	**2**	**4**
Row 2	**6**	**1**
Row 3	**5**	**9**

Product Cell (**1,1**) = A Row 1 x B Col 1 = (4 x 2 + 5 x 6 + 8 x 5) = **78**

Product Cell (**2,1**) = A Row 1 x B Col 2 = (4 x 4 + 5 x 1 + 8 x 9) = **93**

Product Cell (**1,2**) = A Row 2 x B Col 1 = (1 x 2 + 9 x 6 + 7 x 5) = **91**

Product Cell (**2,2**) = A Row 2 x B Col 2 = (1 x 4 + 9 x 1 + 7 x 9) = **76**

Matrix C

(2 rows x 2 cols)

78	**93**
91	**76**

Adjacency Matrix

- If it is **true** you can get from **A to B**, and it is **true** you can get from **B to C**, then it must be **true** that you can get from **A to C** (transitive property)
- When “multiplying” **bool** adjacency matrices, if the **AND** of all the elements in (Row A x Col B) is **== true** then the cell is **set to true**
  - We keep multiplying the adjacency matrix until a **true** appears in the cell that represents the exit square.
  - The total # of matrix multiplications required **= the shortest path length from entrance to exit**

Adjacency Matrix

- We can calculate the adjacency matrix **before** starting a depth-first search. We can then use this **shortest path length** to limit the current search path to improve the efficiency of the search
- Once the current **stack.size()** has more levels than the shortest path length calculated from the adjacency matrix, **start back tracking!**
- There is no reason to continue on a path which has a step count greater than the known shortest path **– it's best to backup and try a new direction**

Lab 3 – Maze Search

Lab 2 – Maze Search w/o Adj Matrix

```
int main()
{
    LoadMaze();
    ResetVisitCount();

    tuple<int, int, int> entrance(0, 0, 1);
    stack.push_back(entrance);
    visitCount[0][0] = 1;

    SimpleScreen ss(draw, eventHandler);
    ss.SetZoomFrame("white", 3);
    ss.SetWorldRect(-10, -10, 460, 460);

    DrawMaze(ss);

    ss.HandleEvents();

    return 0;
}
```

Lab 3 – Maze Search w Adj Matrix

```
int main()
{
    LoadMaze();
    ResetVisitCount();

    AdjMatrix adj;
    adj.Init(maze);
    minSteps = adj.MinSteps();

    tuple<int, int, int> entrance(0, 0, 1);
    stack.push_back(entrance);
    visitCount[0][0] = 1;

    SimpleScreen ss(draw, eventHandler);
    ss.SetZoomFrame("white", 3);
    ss.SetWorldRect(-10, -10, 460, 460);

    DrawMaze(ss);

    ss.HandleEvents();

    return 0;
}
```

The variable minSteps holds the # of steps in the shortest path from entrance to exit

Lab 3 – Maze Search

Lab 2 – Maze Search w/o Adj Matrix

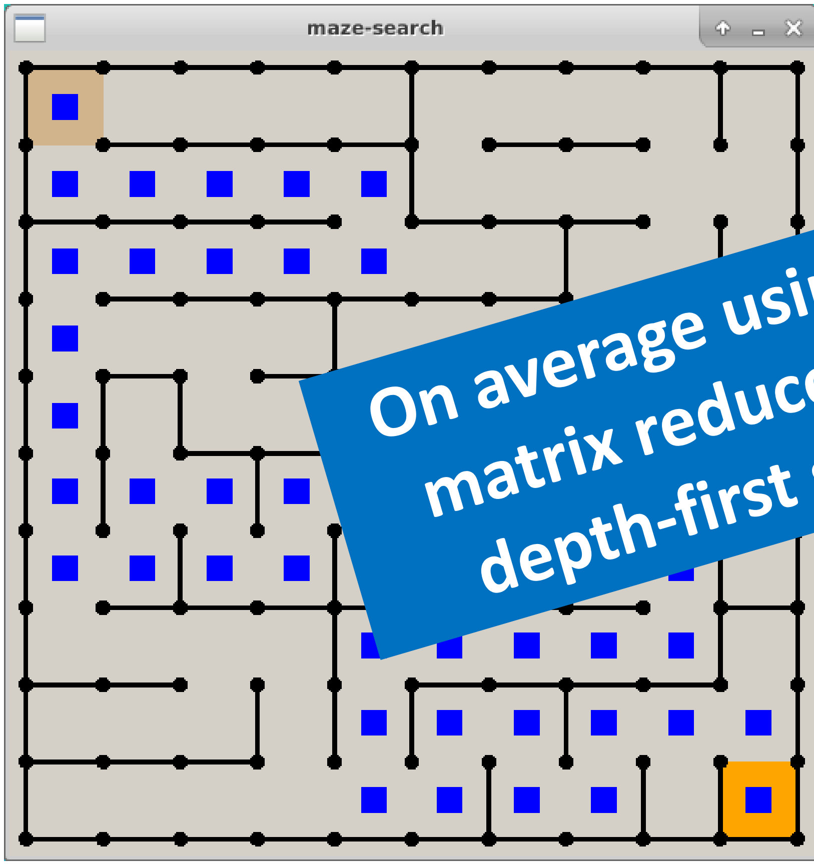
```
bool TakeStep()
{
    int r = get<0>(stack.back());
    int c = get<1>(stack.back());
    int r2 = r; int c2 = c;
    int dir = get<2>(stack.back());
    if (dir == 1) r2--;
    if (dir == 2) c2++;
    if (dir == 4) r2++;
    if (dir == 8) c2--;
    get<2>(stack.back()) *= 2;
    bool moved = false;
    if (((maze[r][c] & dir) != dir) &&
        visitCount[r2][c2] == 0)
    {
        tuple<int, int, int> cell(r2, c2, 1);
        stack.push_back(cell);
        moved = true;
    }
    if (dir == 16 && !moved) {
        stack.pop_back();
        r2 = get<0>(stack.back());
        c2 = get<1>(stack.back());
        moved = true;
    }
    if (moved) {
        visitCount[r2][c2]++;
        totalSteps++;
        if (r2 == 9 && c2 == 9)
            foundExit = true;
        return true;
    }
    return false;
}
```

Lab 3 – Maze Search w Adj Matrix

```
bool TakeStep()
{
    int r = get<0>(stack.back());
    int c = get<1>(stack.back());
    int r2 = r; int c2 = c;
    int dir = get<2>(stack.back());
    if (dir == 1) r2--;
    if (dir == 2) c2++;
    if (dir == 4) r2++;
    if (dir == 8) c2--;
    get<2>(stack.back()) *= 2;
    bool moved = false;
    if (((maze[r][c] & dir) != dir) &&
        (visitCount[r2][c2] == 0) &&
        (stack.size() <= minSteps))
    {
        tuple<int, int, int> cell(r2, c2, 1);
        stack.push_back(cell);
        moved = true;
    }
    if (dir == 16 && !moved) {
        stack.pop_back();
        r2 = get<0>(stack.back());
        c2 = get<1>(stack.back());
        moved = true;
    }
    if (moved) {
        visitCount[r2][c2]++;
        totalSteps++;
        if (r2 == 9 && c2 == 9)
            foundExit = true;
        return true;
    }
    return false;
}
```

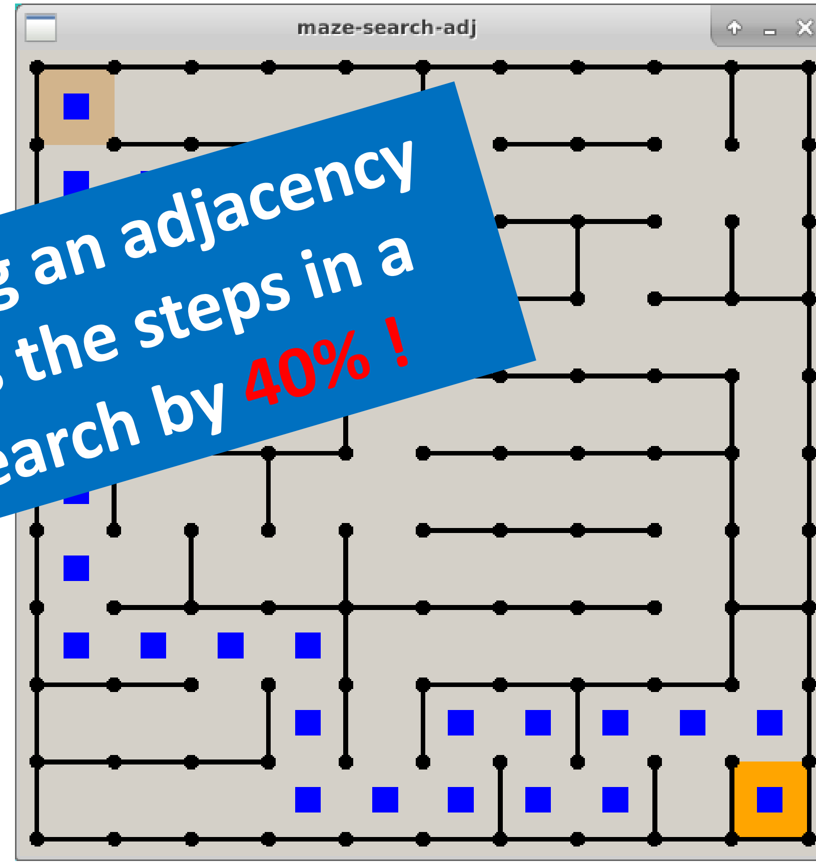
Lab 3 – Maze Search

Lab 2 – Maze Search w/o Adj Matrix



Path Steps = 42

Lab 3 – Maze Search with Adj Matrix



Path Steps = 30

On average using an adjacency matrix reduces the steps in a depth-first search by 40% !

Now you know…

- How to encode 2D maze walls in **base 2**
  - In C++ an **[Y][X]** matrix means there are **Y rows** and **X columns**
  - The bitwise AND (&) operator can decode wall values
- Depth-first search is implemented with recursion or a **stack**
  - You must use a **breadcrumbs** array to prevent infinite loops
- A **logical adjacency matrix** can be used to calculate the length of the shortest path from entrance to exit
  - However, the adjacency matrix will not identify the actual steps along that shortest path
  - Leveraging the adjacency matrix during a depth-first search will yield on average a ~**40% improvement in** the efficiency of the search