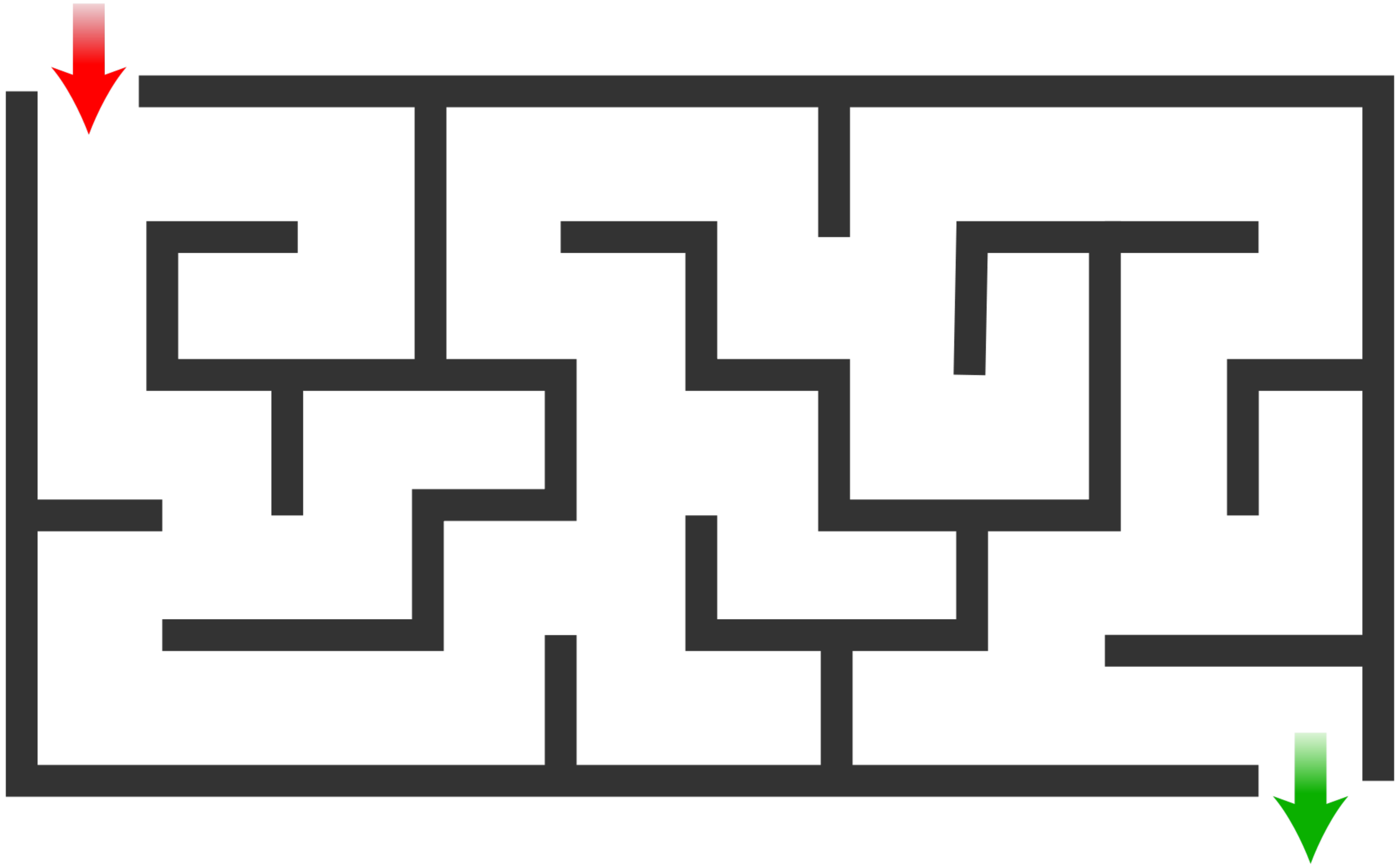
CS 106B

Lecture 10: Recursive Backtracking 2: Common Problem Types

Wednesday, July 12, 2017



Programming Abstractions
Summer 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 8.2-8.3

Today's Topics

- Logistics:
  - Due date for Assignment 3 (Recursion): Tuesday, Noon
  - Practice midterm materials: http://web.stanford.edu/class/cs106b/handouts/midterm.html
  - Midterm will be on laptops! You must let me and Jason know now if you don't have a workable laptop or need other accommodations.
  - Today's handout: http://web.stanford.edu/class/cs106b//lectures/10-RecursiveBacktracking2/code/handout.pdf

- Common Problem Types for Recursive Backtracking
  - Partitionable (determine whether a solution exists)
  - Knapsack Problem (find the best solution)
  - Maze Solving (find a solution)
  - Clumsy Thumbsy (find all solutions)

Recursive Backtracking: Templates 🔑

There are basically five different problems you might see that will require recursive backtracking:

- Determine whether a solution exists
- Find a solution
- Find the best solution
- Count the number of solutions
- Print/find all the solutions

Partitionable: determine whether a solution exists

Write a function named **`partionable`** that takes a vector of `int`s and returns `true` if it is possible to divide the `int`s into two groups such that each group has the same sum. For example, the `Vector {1,1,2,3,5}` can be split into `{1,5}` and `{1,2,3}`. However, the `vector {1,4,5,6}` can't be split into two.

```
bool partitionable(Vector<int>& nums) { ...
```

Partitionable: determine whether a solution exists

```
bool partitionable(Vector<int>& nums) { ...
```

This is our first example of recursive backtracking where we **make a change and must restore some data before we can move on**; otherwise, the solution degrades.

Basic idea:

- Keep track of the two sums! Must use helper function.
- Keep removing values from the Vector until we have no more values left (base case)
- Search each possible path

```
bool partitionable(Vector<int>& rest, int sum1, int sum2);
```

Partitionable: determine whether a solution exists

```
bool partitionable(Vector<int>& nums) {
    return partitionable(nums, 0, 0); // no sums yet
}

bool partitionable(Vector<int>& rest, int sum1, int sum2) {



}
```

Partitionable: determine whether a solution exists

```
bool partitionable(Vector<int>& nums) {
    return partitionable(nums, 0, 0); // no sums yet
}

bool partitionable(Vector<int>& rest, int sum1, int sum2) {
    if (rest.isEmpty()) {
        return sum1 == sum2;
    } else {
        int n = rest[0];
        rest.remove(0);
        bool answer = partitionable(rest, sum1 + n, sum2)
                   || partitionable(rest, sum1, sum2 + n);
        rest.insert(0, n);
        return answer;
    }
}
```

base case: note the return value

adjust rest (must restore!!!)

here is the restoration

The Knapsack Problem: Find the best solution

One famous problem in theoretical computer science is the so-called *knapsack problem*. Given a target weight and a set of objects in which each object has a value and a weight, determine a subset of objects such that the sum of their weights is less than or equal to the target weight and the sum of their values is maximized.

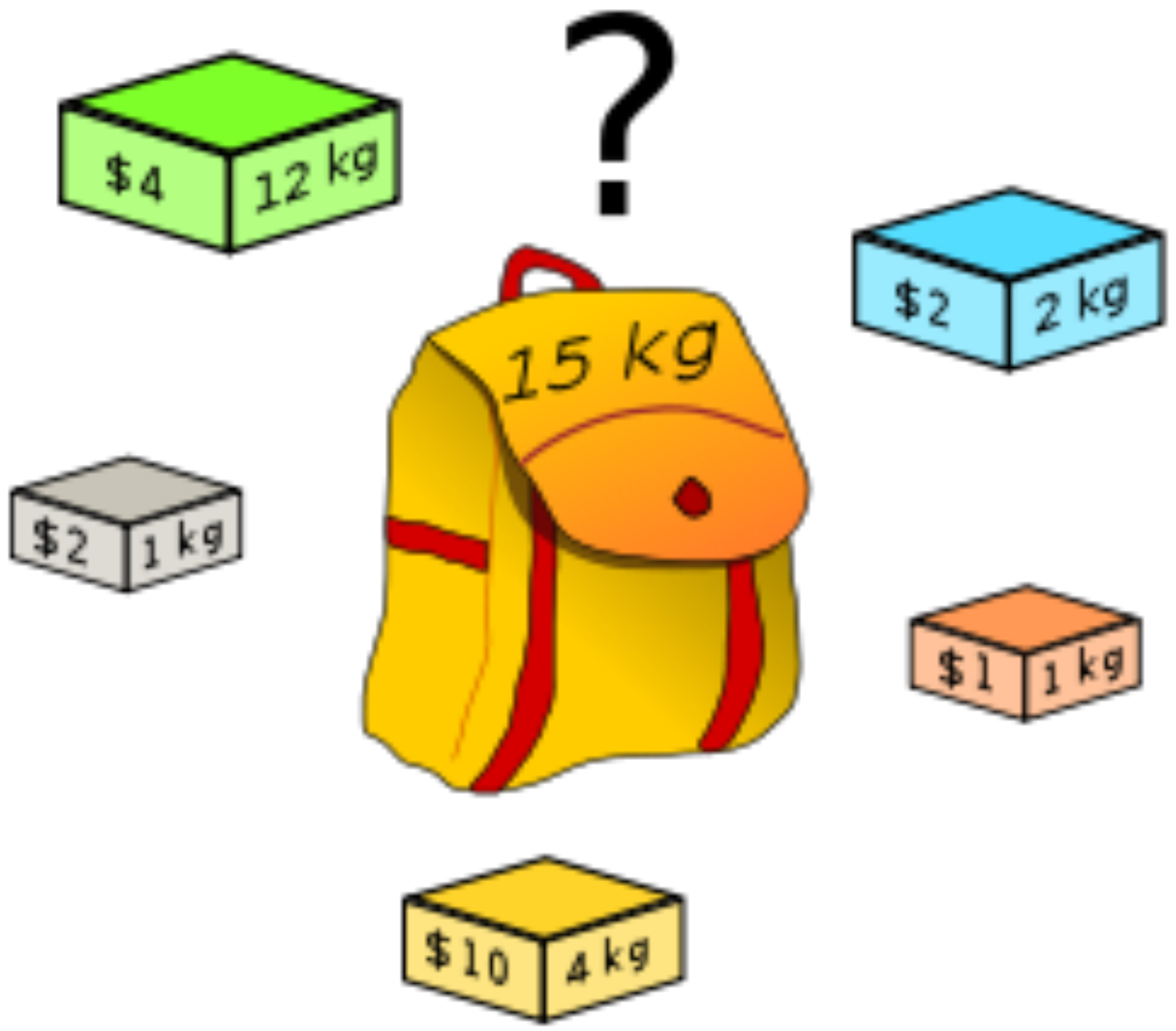


image courtesy of wikipedia.org

The Knapsack Problem: Find the best solution

For this problem we will represent an object with the following struct:

```
struct objectT {
    int weight; //You may assume this is greater than or equal to 0
    int value;  //You may assume this is greater than or equal to 0
};
```

Let's write the function:

```
int fillKnapsack(Vector<objectT> &objects, int targetWeight)
```

that considers all possible combinations of `objectT` from objects (such that the sum of their weights is less than or equal to targetWeight) and returns the maximum possible sum of object values.

The Knapsack Problem: Find the best solution

```
int fillKnapsack(Vector<objectT> &objects, int targetWeight)
```

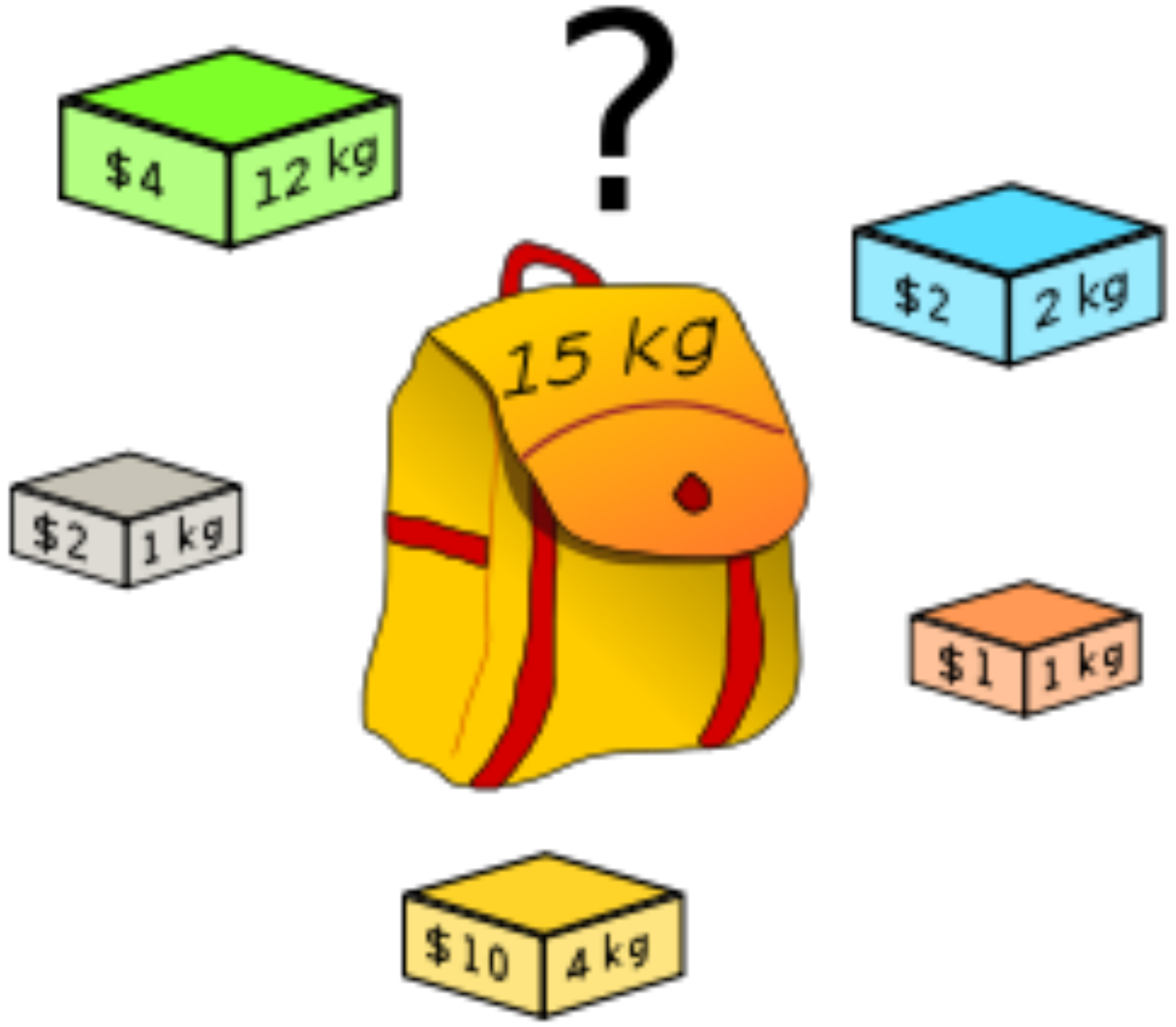


image courtesy of wikipedia.org

Basic idea:

- Keep track of the weight and keep track of the best total value ("score").
- Loop over all items, adding value to the knapsack, and subtracting the weight of items from the total weight allowed.
- If the weight goes below zero, we have too many items.
- Must have a helper function!

```
int fillKnapsack(Vector<objectT> &objects, int weight, int bestScore);
```

The Knapsack Problem: Solution

Setup struct and call to recursive function:

```
struct objectT {
    int weight; //You may assume this is greater than or equal to 0
    int value;  //You may assume this is greater than or equal to 0
};


int fillKnapsack(Vector<objectT> &objects, int targetWeight) {
    return fillKnapsack(objects,targetWeight,0);
}
```

The Knapsack Problem: Solution

```
int fillKnapsack(Vector<objectT> &objects, int weight, int bestScore) {
    if (weight < 0) return 0; // we tried too much weight!
```

base case

The Knapsack Problem: Solution

```
int fillKnapsack(Vector<objectT> &objects, int weight, int bestScore) {
    if (weight < 0) return 0; // we tried too much weight!
    int localBestScore = bestScore;
```

local variable to keep track of score

The Knapsack Problem: Solution

```
int fillKnapsack(Vector<objectT> &objects, int weight, int bestScore) {
    if (weight < 0) return 0; // we tried too much weight!
    int localBestScore = bestScore;
    int obSize = objects.size();
    for (int i = 0; i < obSize; i++) {
        objectT originalObject = objects[i];
        int currValue = bestScore + originalObject.value;
        int currWeight = weight - originalObject.weight;
```

loop over all objects, updating the local value and weight

The Knapsack Problem: Solution

```
int fillKnapsack(Vector<objectT> &objects, int weight, int bestScore) {
    if (weight < 0) return 0; // we tried too much weight!
    int localBestScore = bestScore;
    int obSize = objects.size();
    for (int i = 0; i < obSize; i++) {
        objectT originalObject = objects[i];
        int currValue = bestScore + originalObject.value;
        int currWeight = weight - originalObject.weight;
        // remove object for recursion
        objects.remove(i);
```

remove the object we are looking at so we can recurse.

The Knapsack Problem: Solution

```
int fillKnapsack(Vector<objectT> &objects, int weight, int bestScore) {
    if (weight < 0) return 0; // we tried too much weight!
    int localBestScore = bestScore;
    int obSize = objects.size();
    for (int i = 0; i < obSize; i++) {
        objectT originalObject = objects[i];
        int currValue = bestScore + originalObject.value;
        int currWeight = weight - originalObject.weight;
        // remove object for recursion
        objects.remove(i);




        // replace
        objects.insert(i,originalObject);
```

remove the object we are looking at so we can recurse. Must remember to replace it!

The Knapsack Problem: Solution

```
int fillKnapsack(Vector<objectT> &objects, int weight, int bestScore) {
    if (weight < 0) return 0; // we tried too much weight!
    int localBestScore = bestScore;
    int obSize = objects.size();
    for (int i = 0; i < obSize; i++) {
        objectT originalObject = objects[i];
        int currValue = bestScore + originalObject.value;
        int currWeight = weight - originalObject.weight;
        // remove object for recursion
        objects.remove(i);
        currValue = fillKnapsack(objects,currWeight,currValue);
        if (localBestScore < currValue) {
            localBestScore = currValue;
        }
        // replace
        objects.insert(i,originalObject);
```

remove the object we are looking at so we can recurse. Must remember to replace it!

The Knapsack Problem: Solution

```
int fillKnapsack(Vector<objectT> &objects, int weight, int bestScore) {
    if (weight < 0) return 0; // we tried too much weight!
    int localBestScore = bestScore;
    int obSize = objects.size();
    for (int i = 0; i < obSize; i++) {
        objectT originalObject = objects[i];
        int currValue = bestScore + originalObject.value;
        int currWeight = weight - originalObject.weight;
        // remove object for recursion
        objects.remove(i);
        currValue = fillKnapsack(objects,currWeight,currValue);
        if (localBestScore < currValue) {
            localBestScore = currValue;
        }
        // replace
        objects.insert(i,originalObject);
    }
    return localBestScore;
}
```

← we return the local best score

Maze Solving: Find a Solution

- A classic example of backtracking is solving a maze: if you go down one path and it isn't the correct path, then you backtrack to your last decision point to try an alternate path.
- If you are using an object passed by reference you need to either *undo* (or "un-choose") paths that fail, or somehow mark them in your object.
- For a maze, you don't want to try and traverse the same path twice, so you need to mark whether you have been down that path before.

Billy Mays Maize Maze

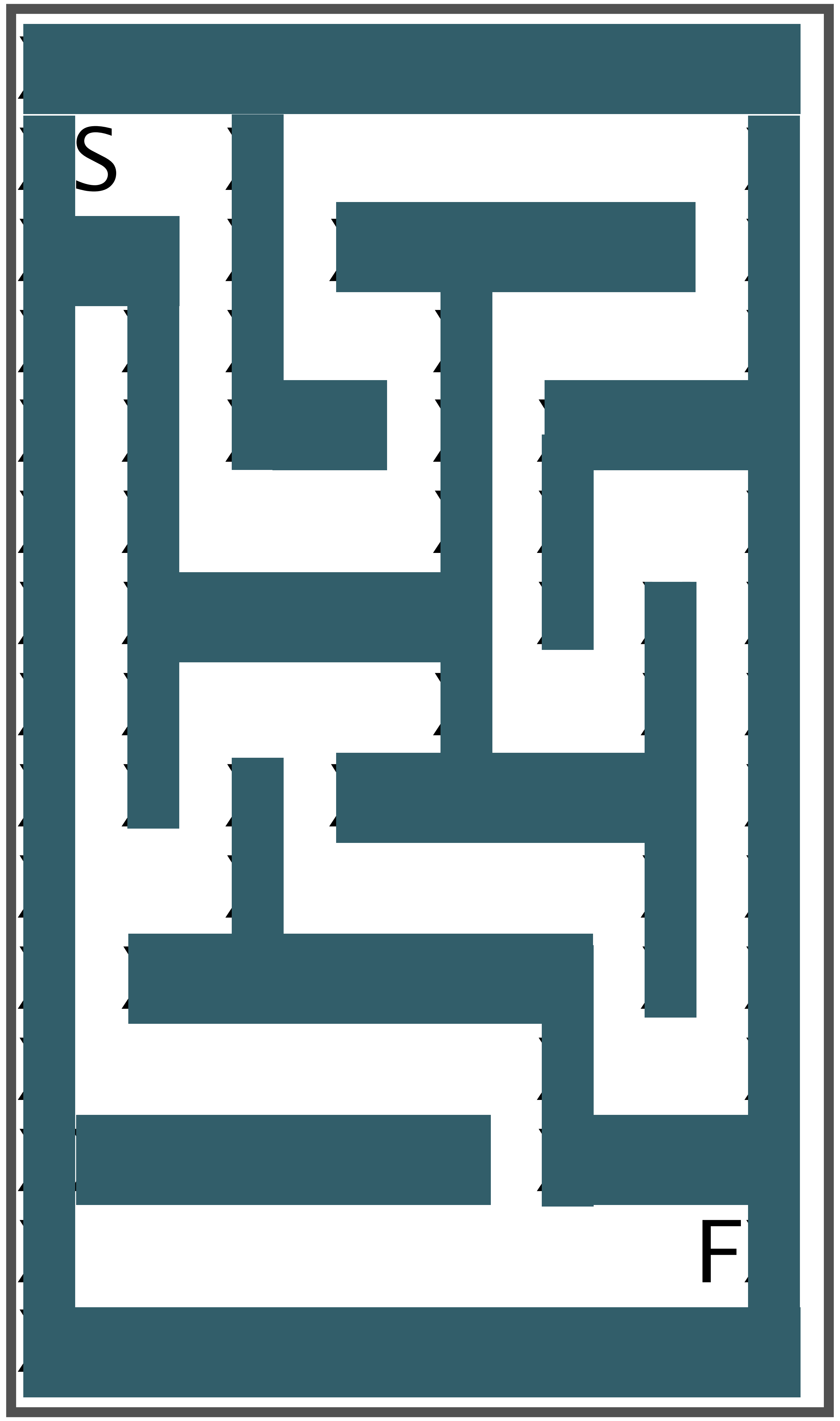
Maze Solving

- The code for today's class includes a text-based recursive maze creator and solver.
- The mazes look like the one to the right
  - There is a Start (marked with an "S") and a Finish (marked with an "F").
  - The Xs represent walls, and the spaces represent paths to walk through the maze.

```
XXXXXXXXXXXXXXX
XS  X         X
XXX X XXXXXXX X
X X X   X     X
X X XXX X XXXXX
X X     X X   X
X XXXXXXX X X X
X X     X   X X
X X X XXXXXXX X
X   X       X X
X XXXXXXXXX X X
X         X   X
XXXXXXXXX XXXXX
X            FX
XXXXXXXXXXXXXXX
```

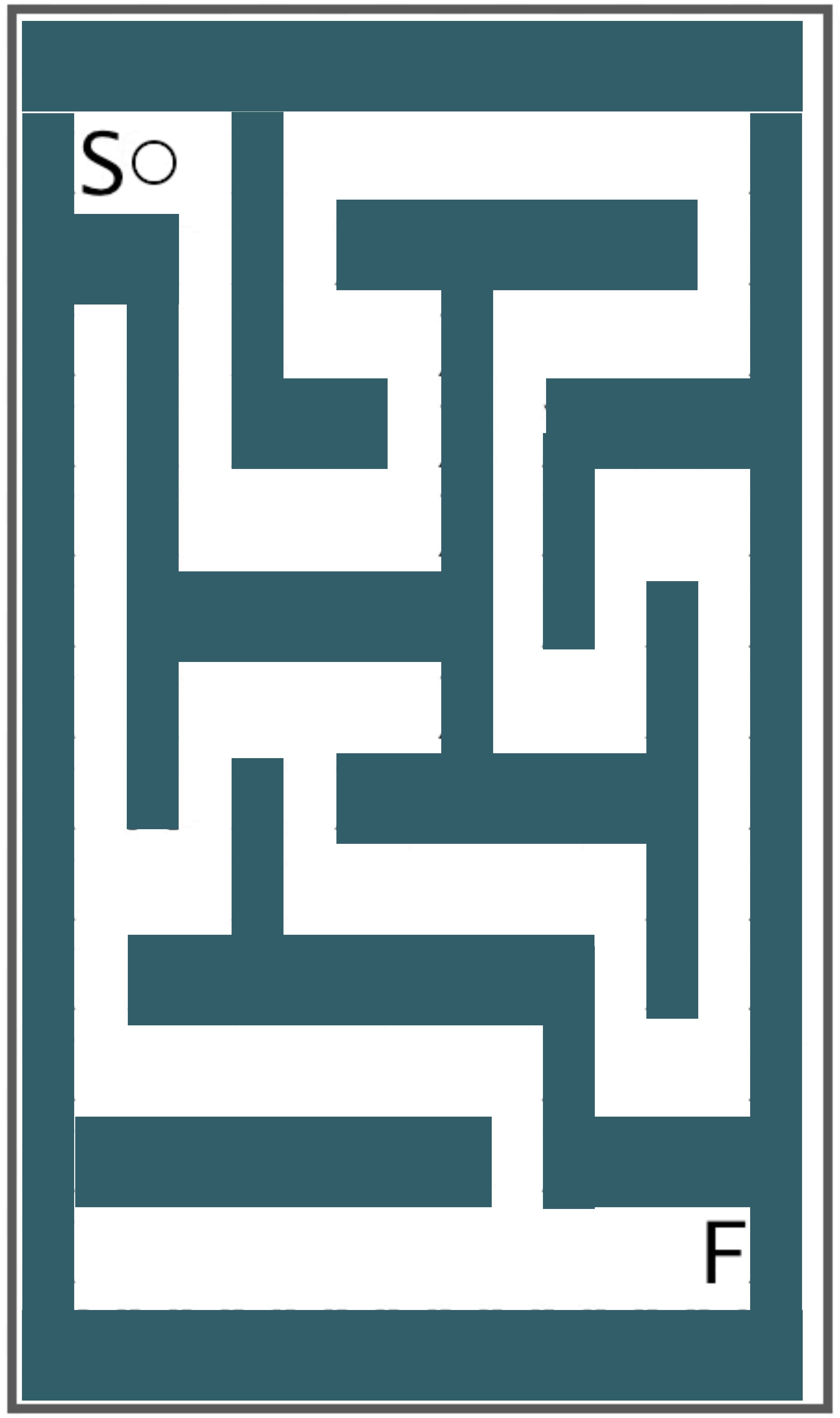
Maze Solving

- Let's make it a bit easier to see on the screen:

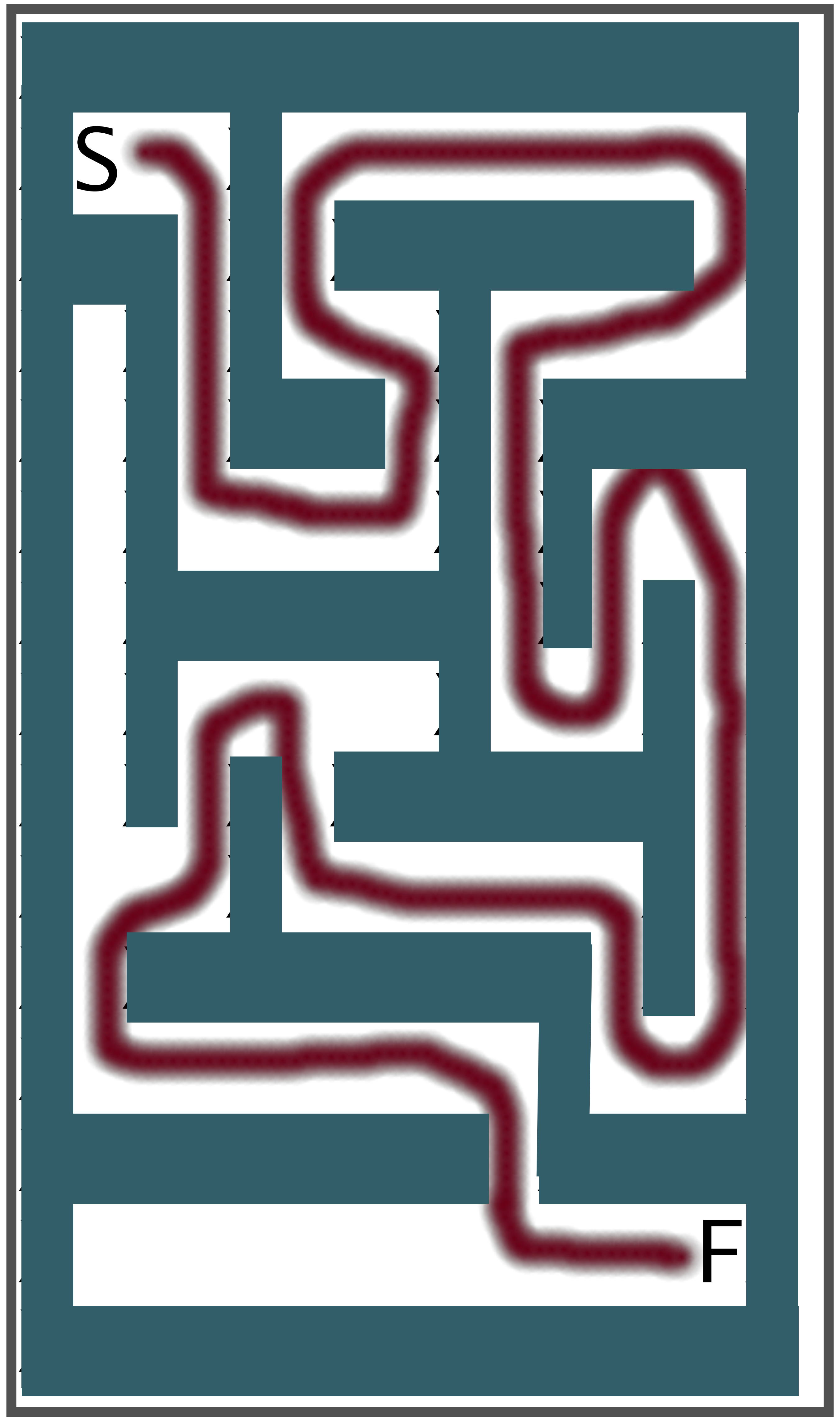


Maze Solving

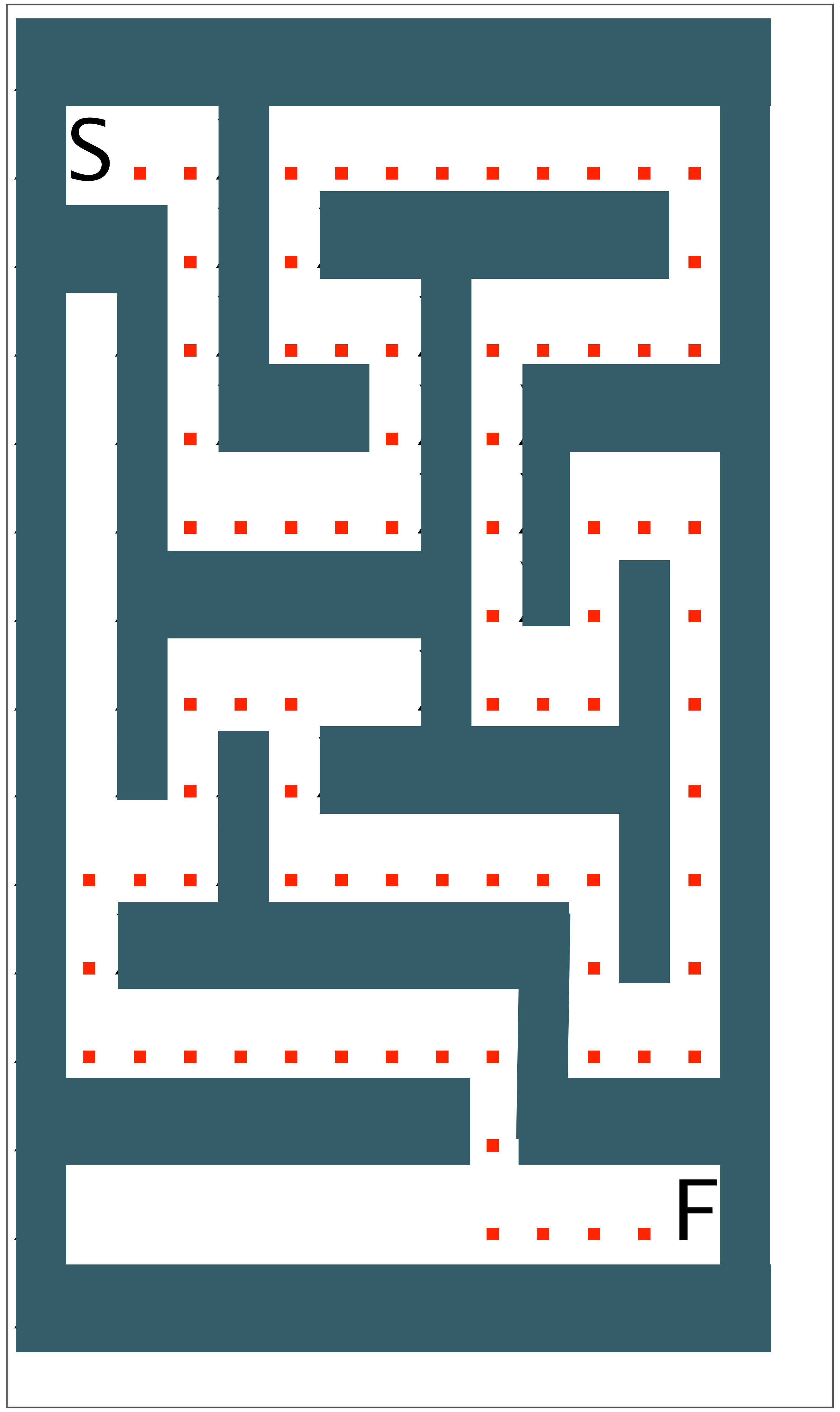
- The solution to the maze is shown here (video):



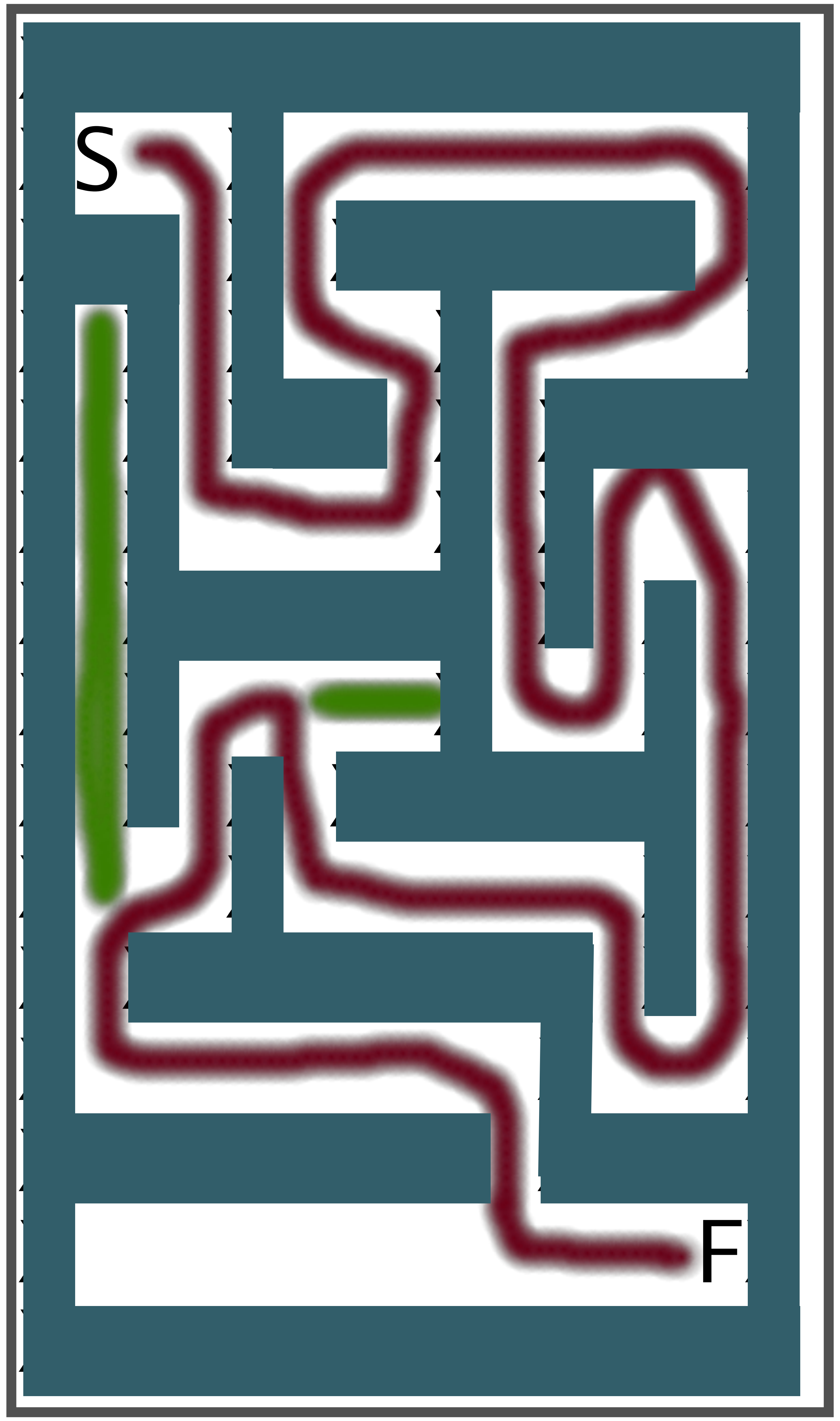
Maze Solving



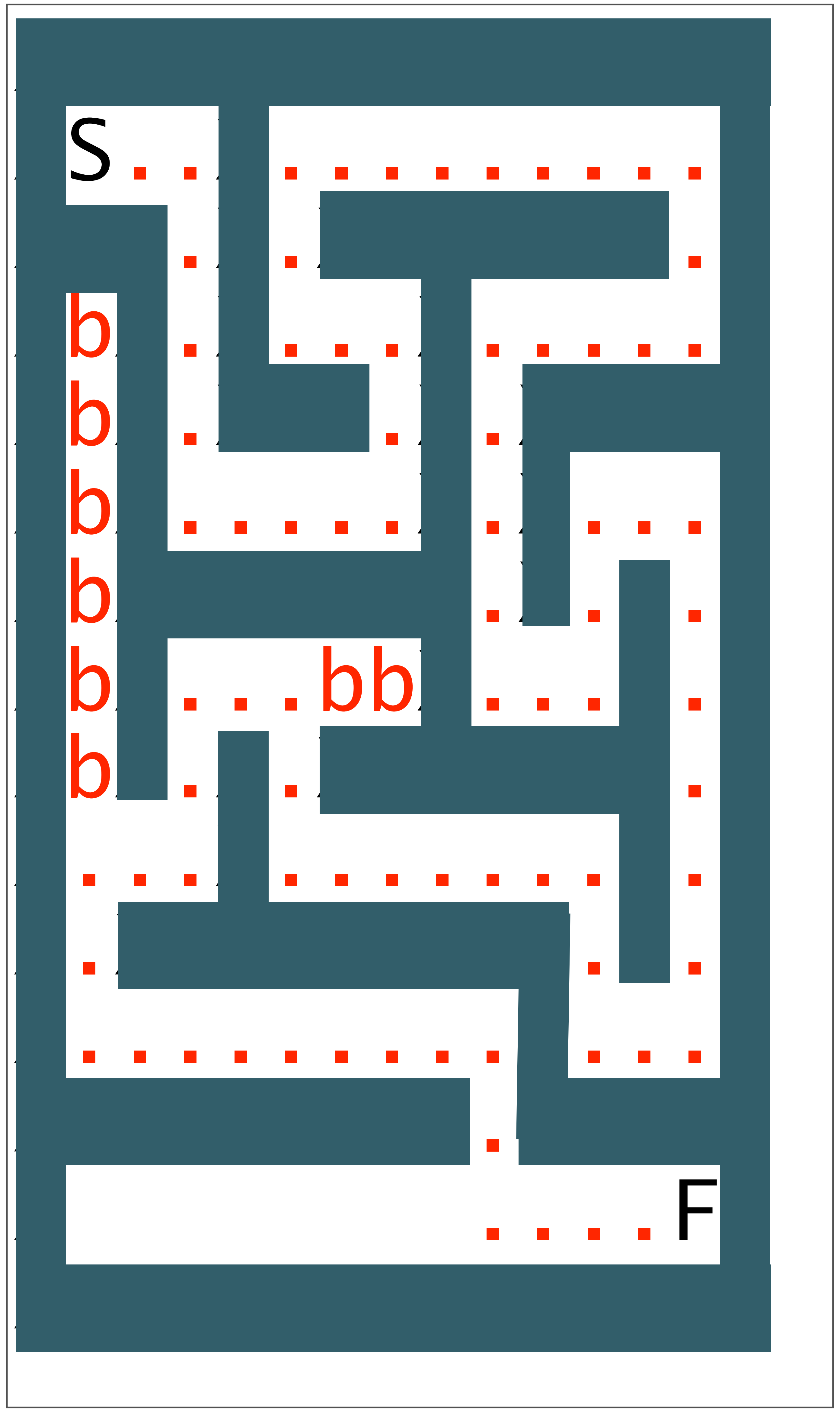
- The program will put dots in the correct positions.



Maze Solving

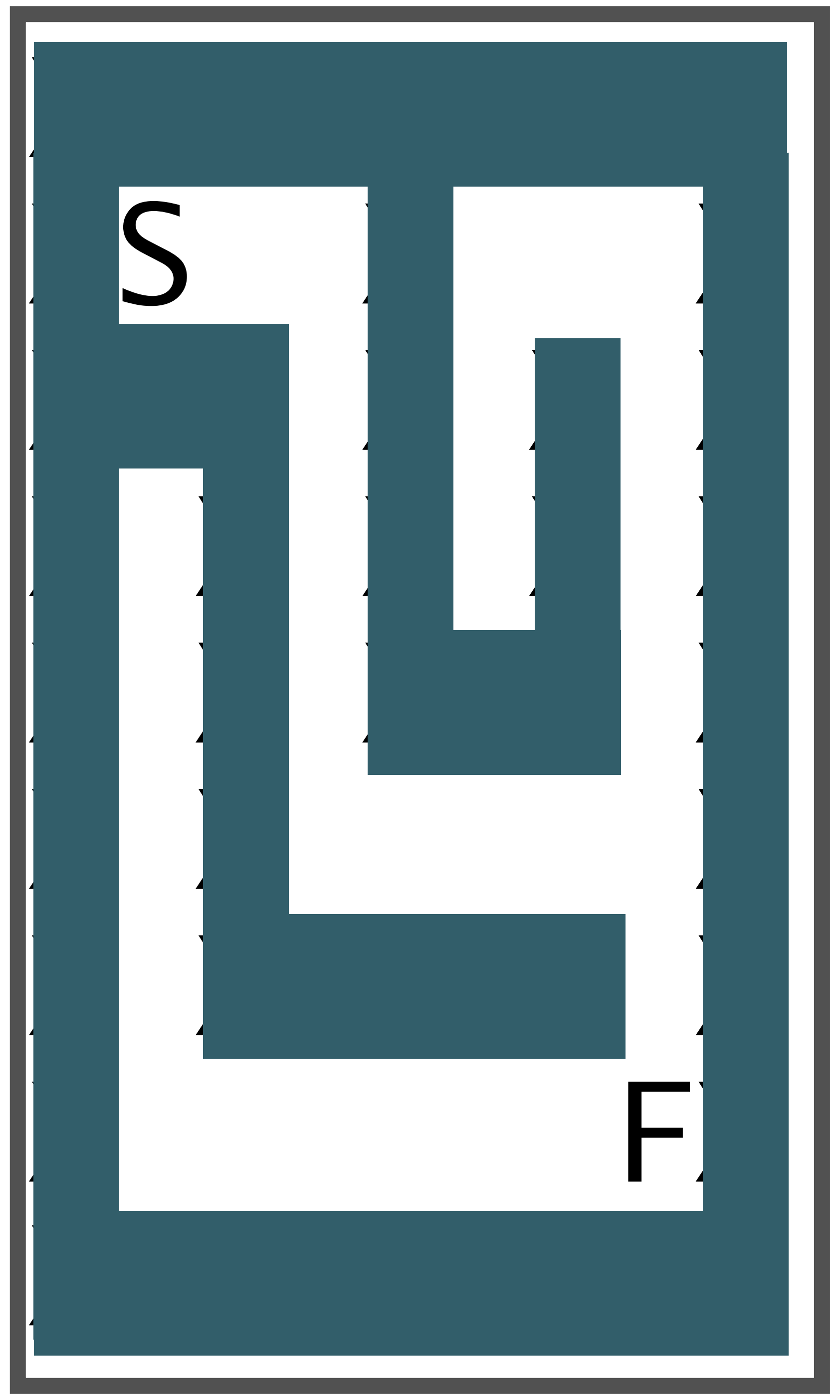


- The program will put dots in the correct positions.
- But, it will also put lowercase b's when it goes in the wrong direction and has to backtrack.



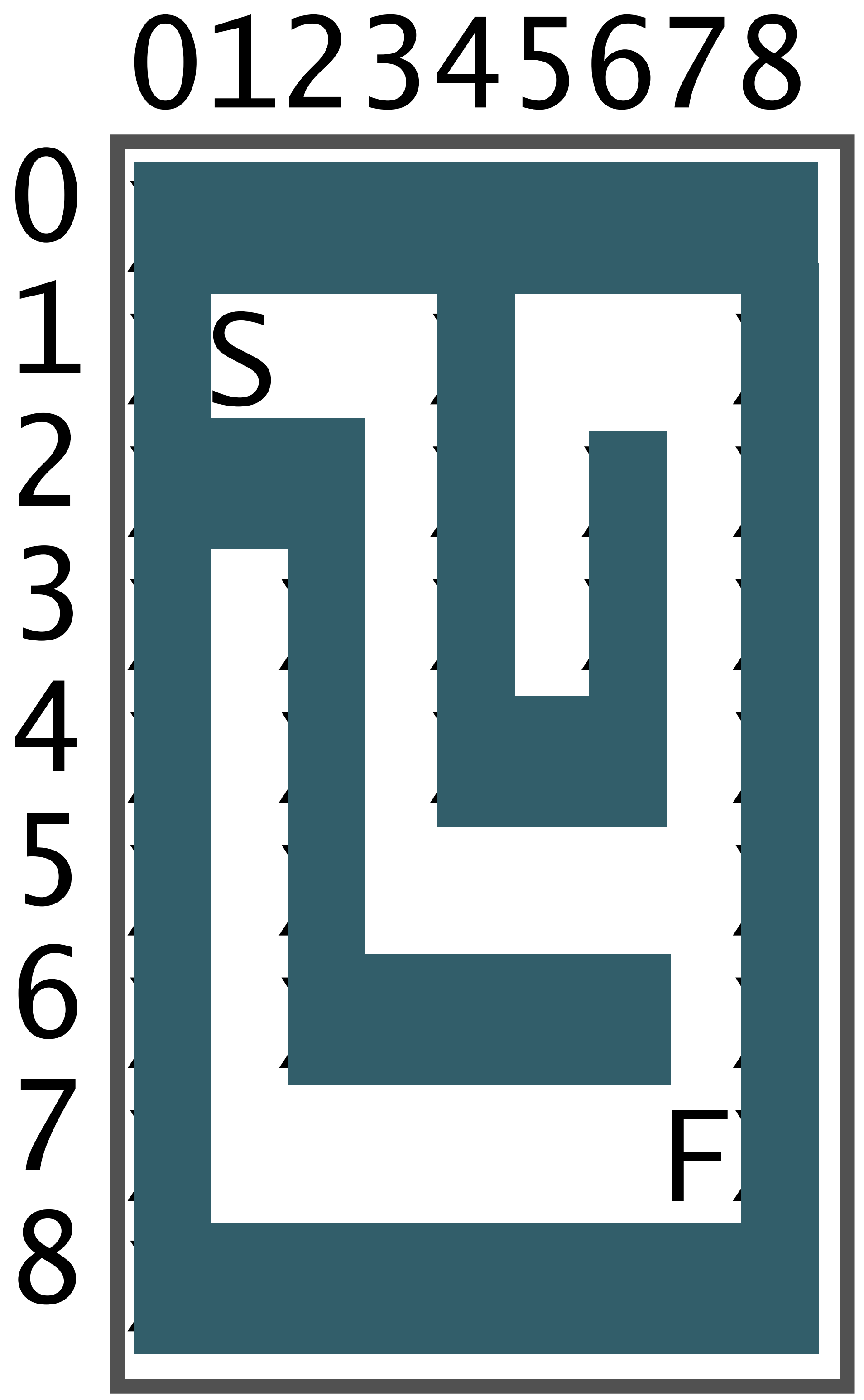
Maze Solving

- What are some actual methods for solving a maze?
  - "Hand on a wall" -- put one hand on a wall at the start and keep following. Eventually you will reach the finish (circular paths may disrupt this method).
  - Break through walls (best for Corn Mazes)
  - Backtracking! Keep track of where you've been, and **systematically test all solutions**. Pick compass directions in order (e.g., N/E/S/W), returning to check other paths when you hit dead ends and have tried all combinations.
- Let's use the backtracking method to solve the maze to the right -- we will go N/E/S/W, from the Start.



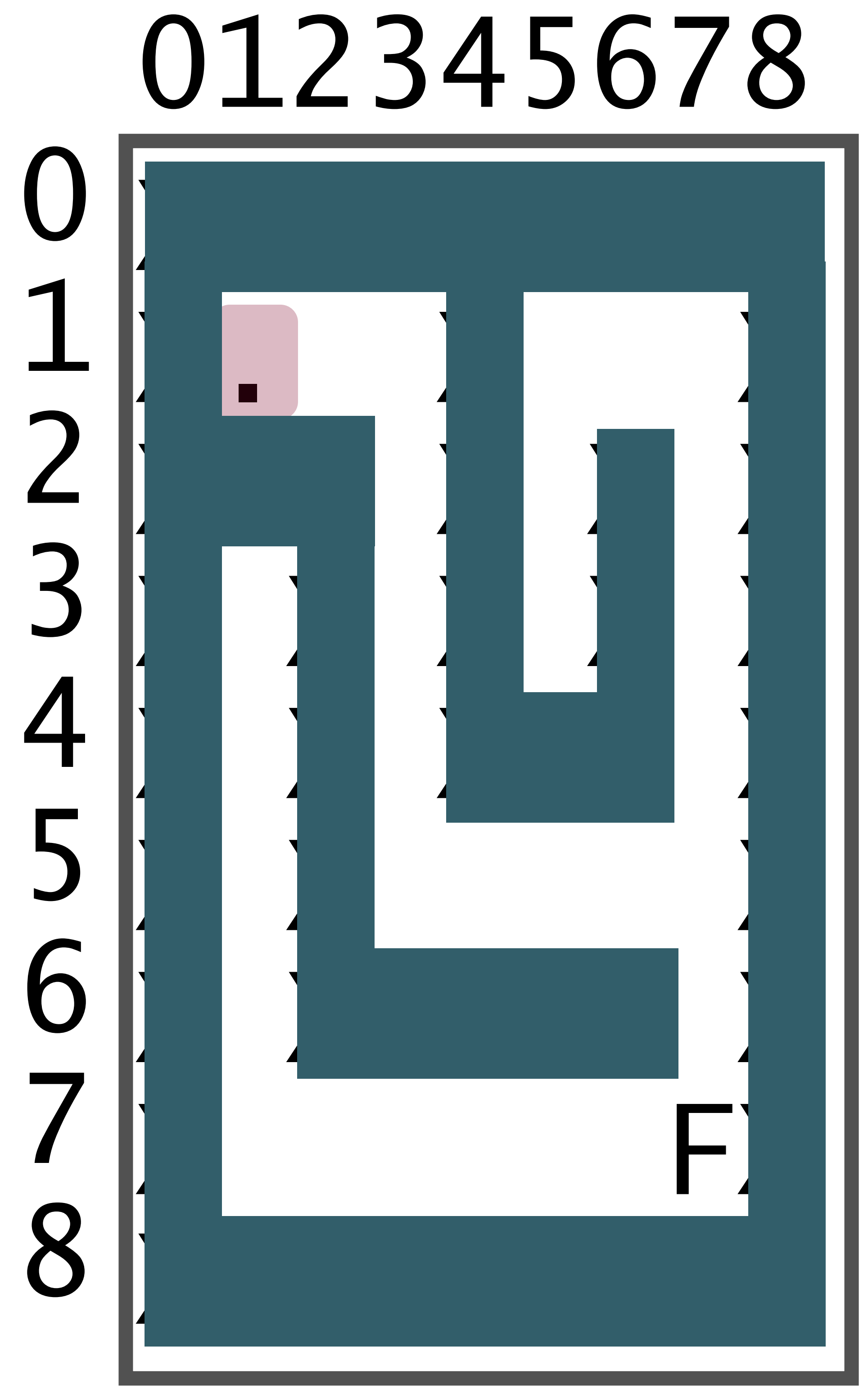
Maze Solving

- We will mark positions we have seen with a period ('.'), and mark backtracking with 'b'.



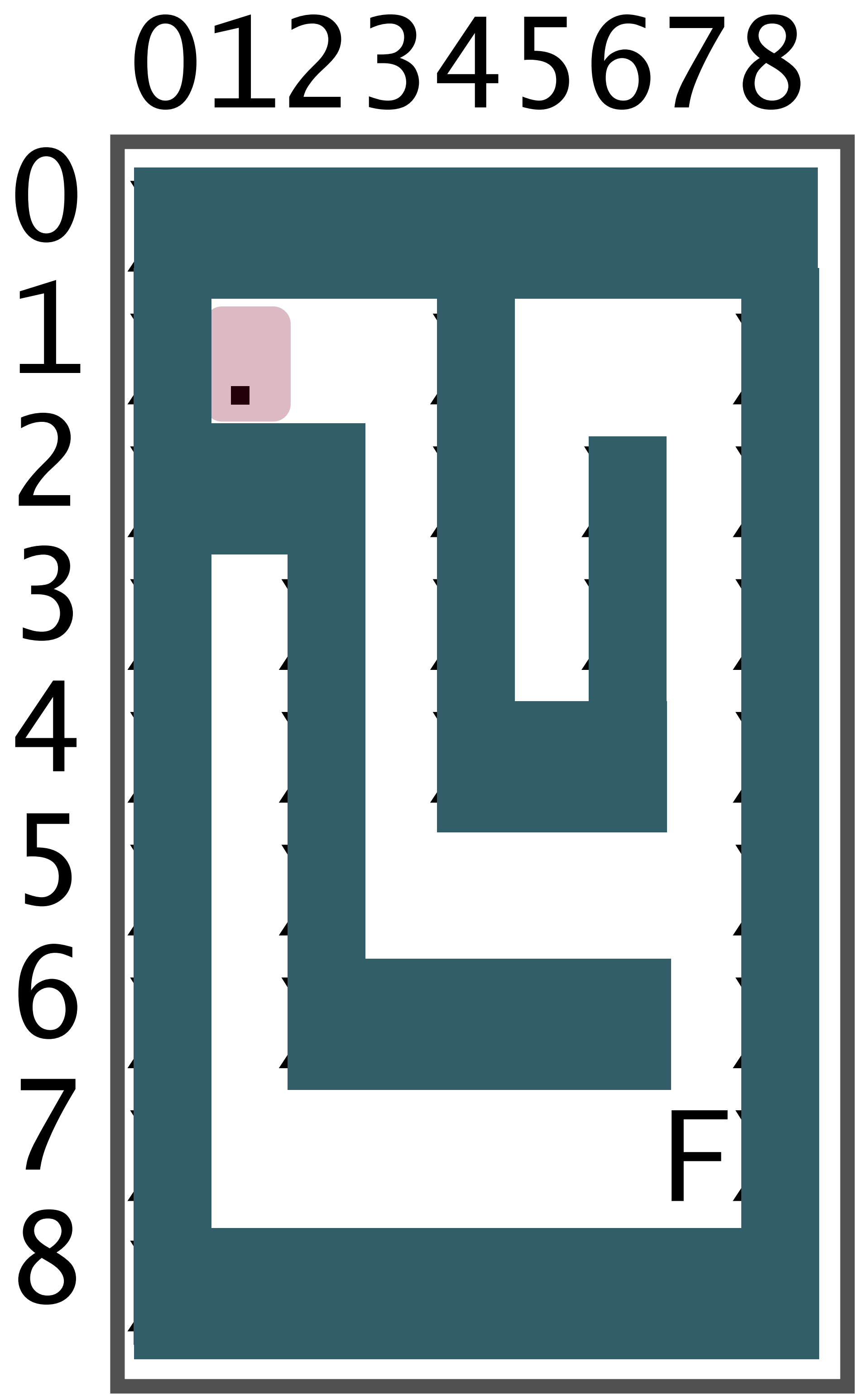
Maze Solving

- We will mark positions we have seen with a period ('.'), and mark backtracking with 'b'.
- **Start: row=1 and col=1, Marking with period (.)**



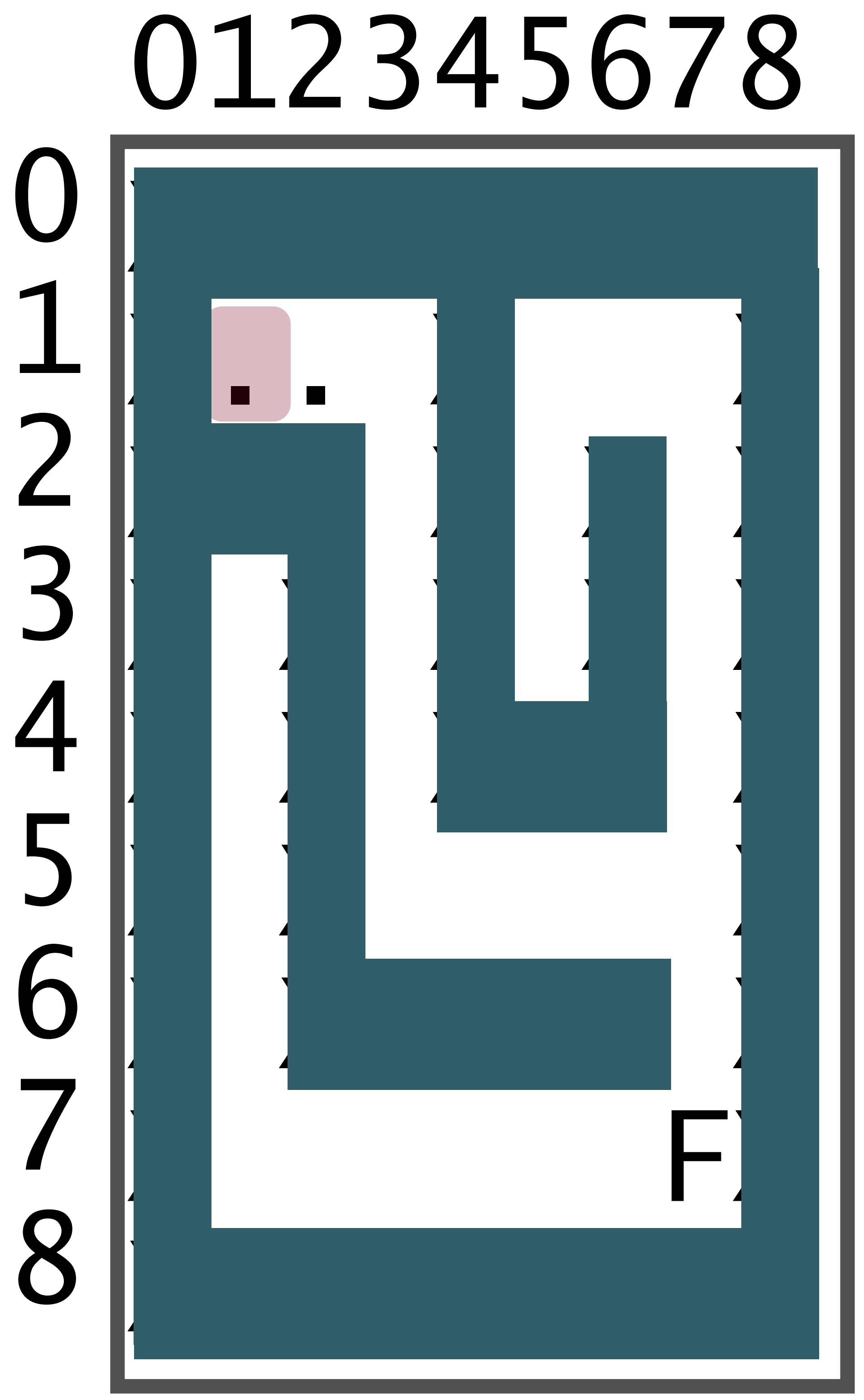
Maze Solving

- We will mark positions we have seen with a period ('.'), and mark backtracking with 'b'.
- Start: row=1 and col=1, Marking with period (.)
- **We have to try all paths, N/E/S/W, and if we hit a wall ('X'), we can't go that direction.**
- **Trying north, row=0 and col=1, Hit wall! Back at row=1 and col=1,**
- **Trying east, row=1 and col=2, Marking with period (.)**



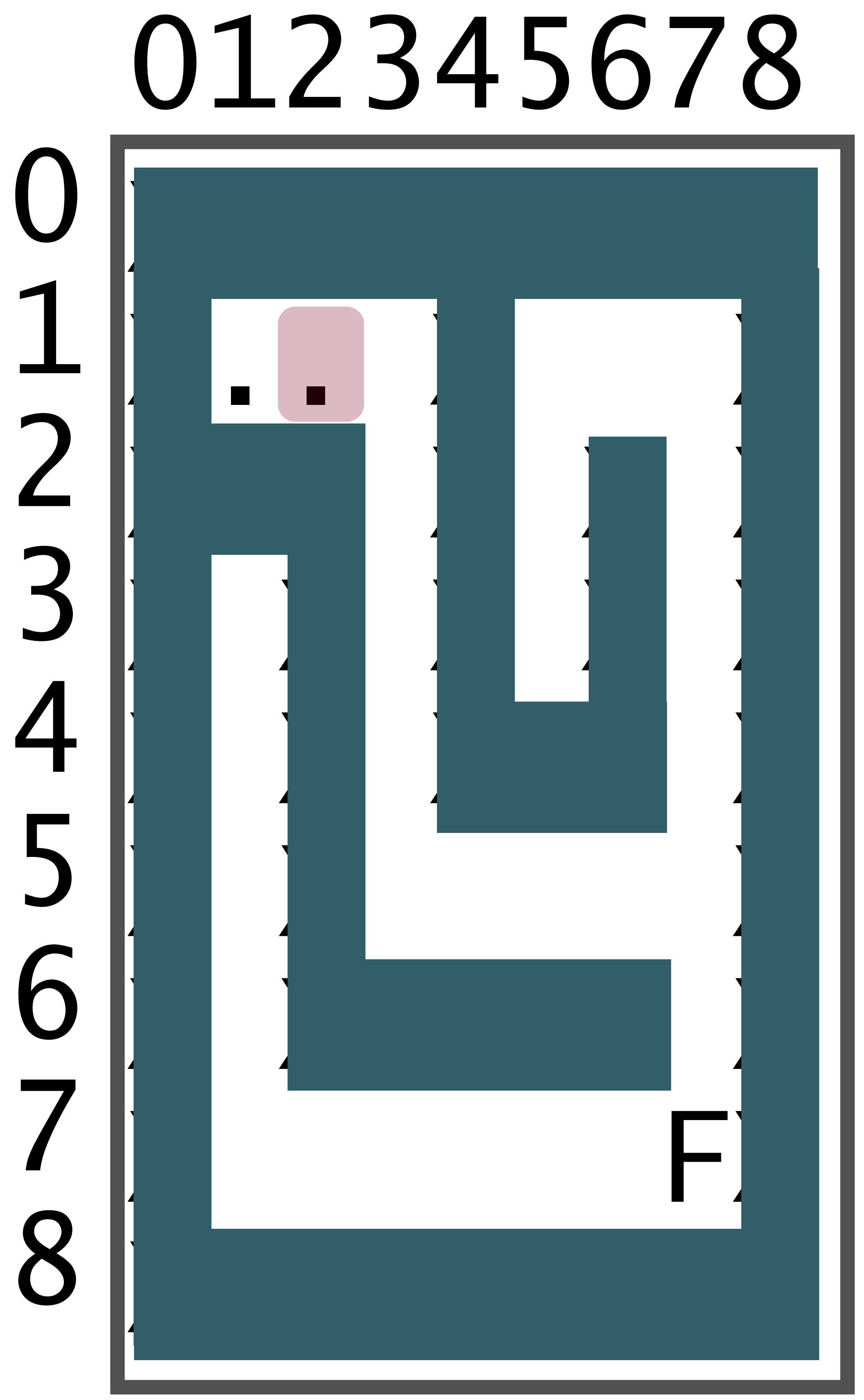
Maze Solving

- We will mark positions we have seen with a period ('.'), and mark backtracking with 'b'.
- Start: row=1 and col=1, Marking with period (.)
- **We have to try all paths, N/E/S/W, and if we hit a wall ('X'), we can't go that direction.**
- **Trying north, row=0 and col=1, Hit wall! Back at row=1 and col=1,**
- **Trying east, row=1 and col=2, Marking with period (.)**



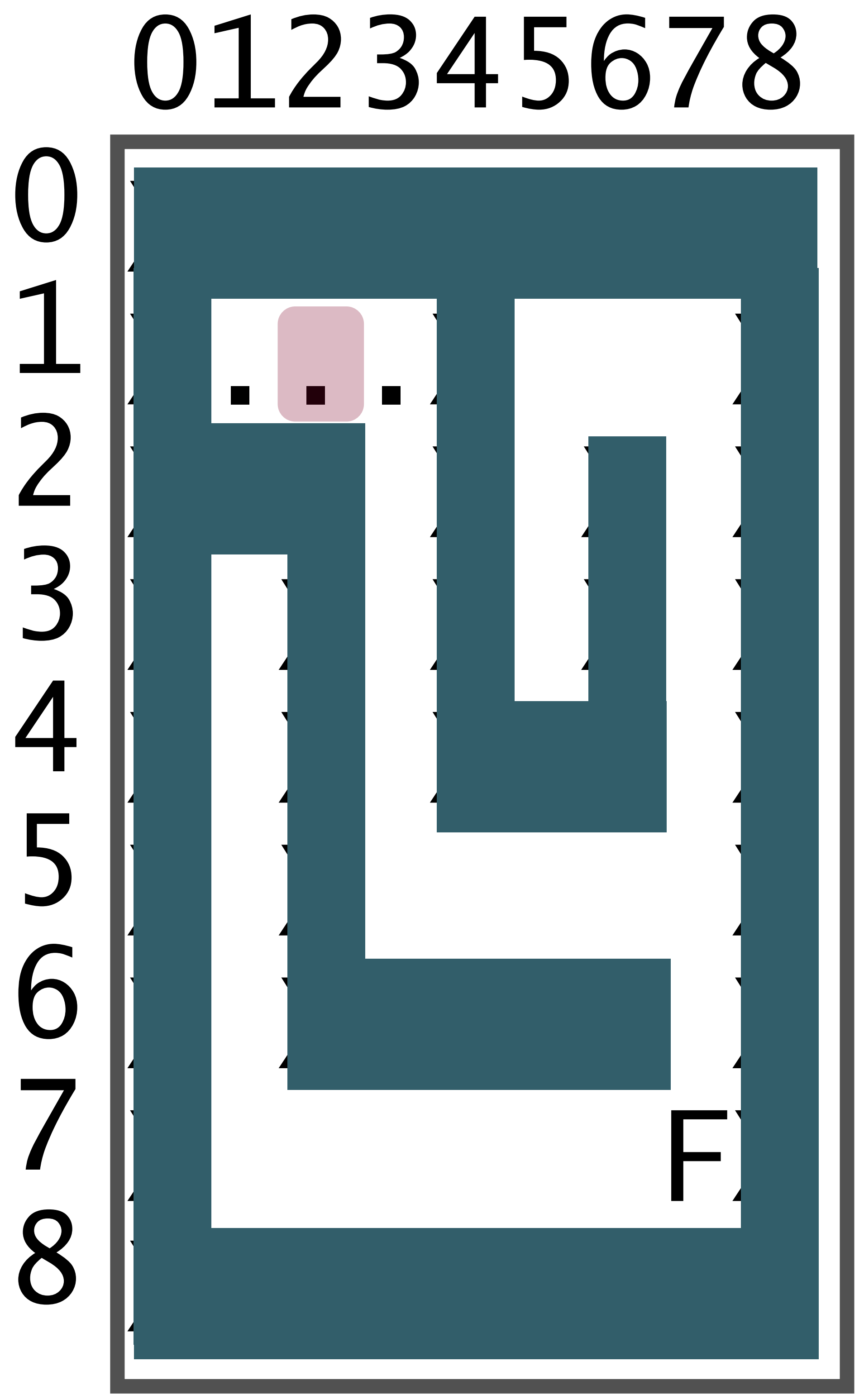
Maze Solving

- We will mark positions we have seen with a period ('.'), and mark backtracking with 'b'.
- Start: row=1 and col=1, Marking with period (.)
- We have to try all paths, N/E/S/W, and if we hit a wall ('X'), we can't go that direction.
- Trying north, row=0 and col=1, Hit wall! Back at row=1 and col=1,
- Trying east, row=1 and col=2, Marking with period (.)
- **Trying north, row=0 and col=2, Hit wall! Back at row=1 and col=2,**
- **Trying east, row=1 and col=3, Marking with period (.)**



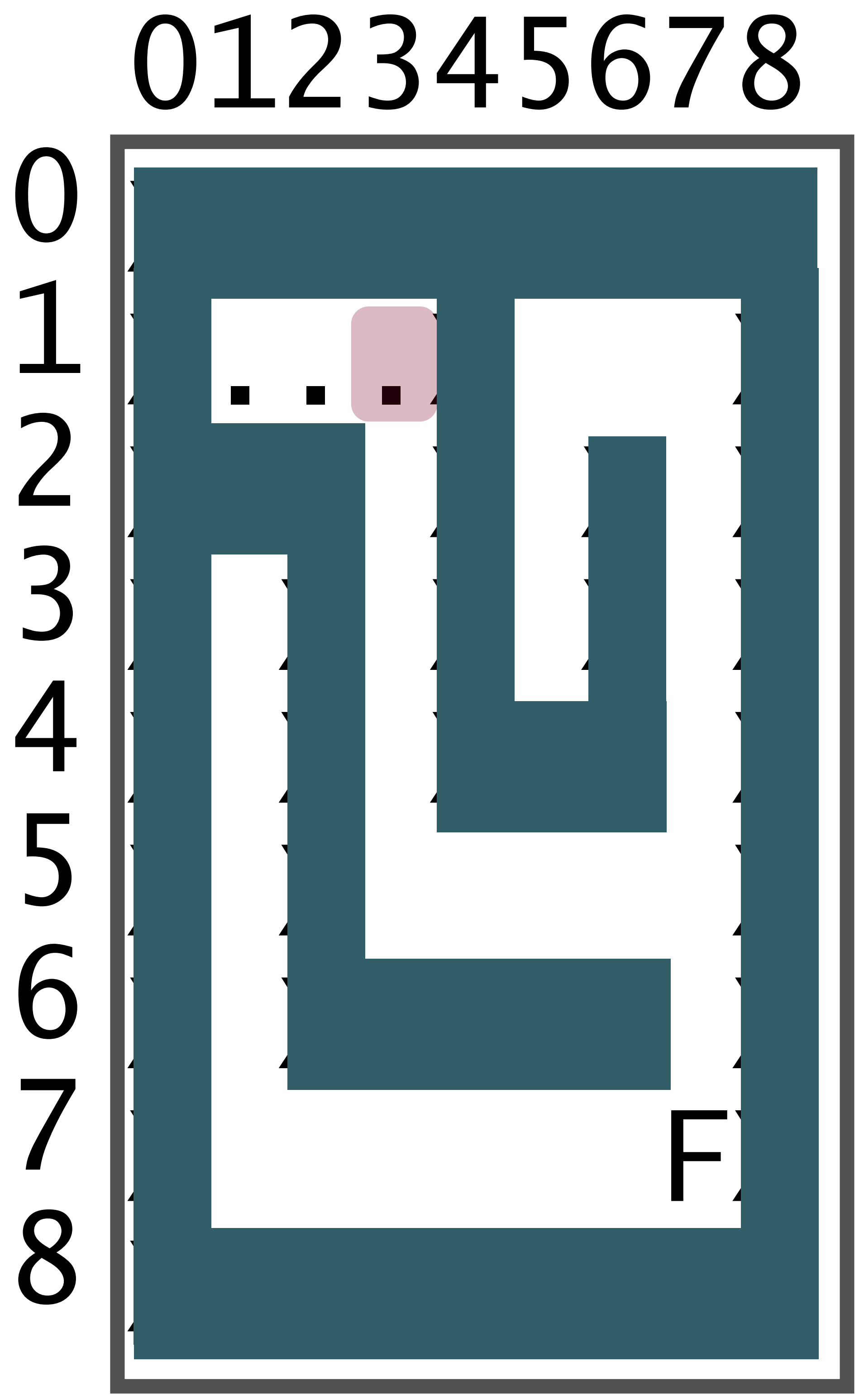
Maze Solving

- We will mark positions we have seen with a period ('.'), and mark backtracking with 'b'.
- Start: row=1 and col=1, Marking with period (.)
- We have to try all paths, N/E/S/W, and if we hit a wall ('X'), we can't go that direction.
- Trying north, row=0 and col=1, Hit wall! Back at row=1 and col=1,
- Trying east, row=1 and col=2, Marking with period (.)
- **Trying north, row=0 and col=2, Hit wall! Back at row=1 and col=2,**
- **Trying east, row=1 and col=3, Marking with period (.)**



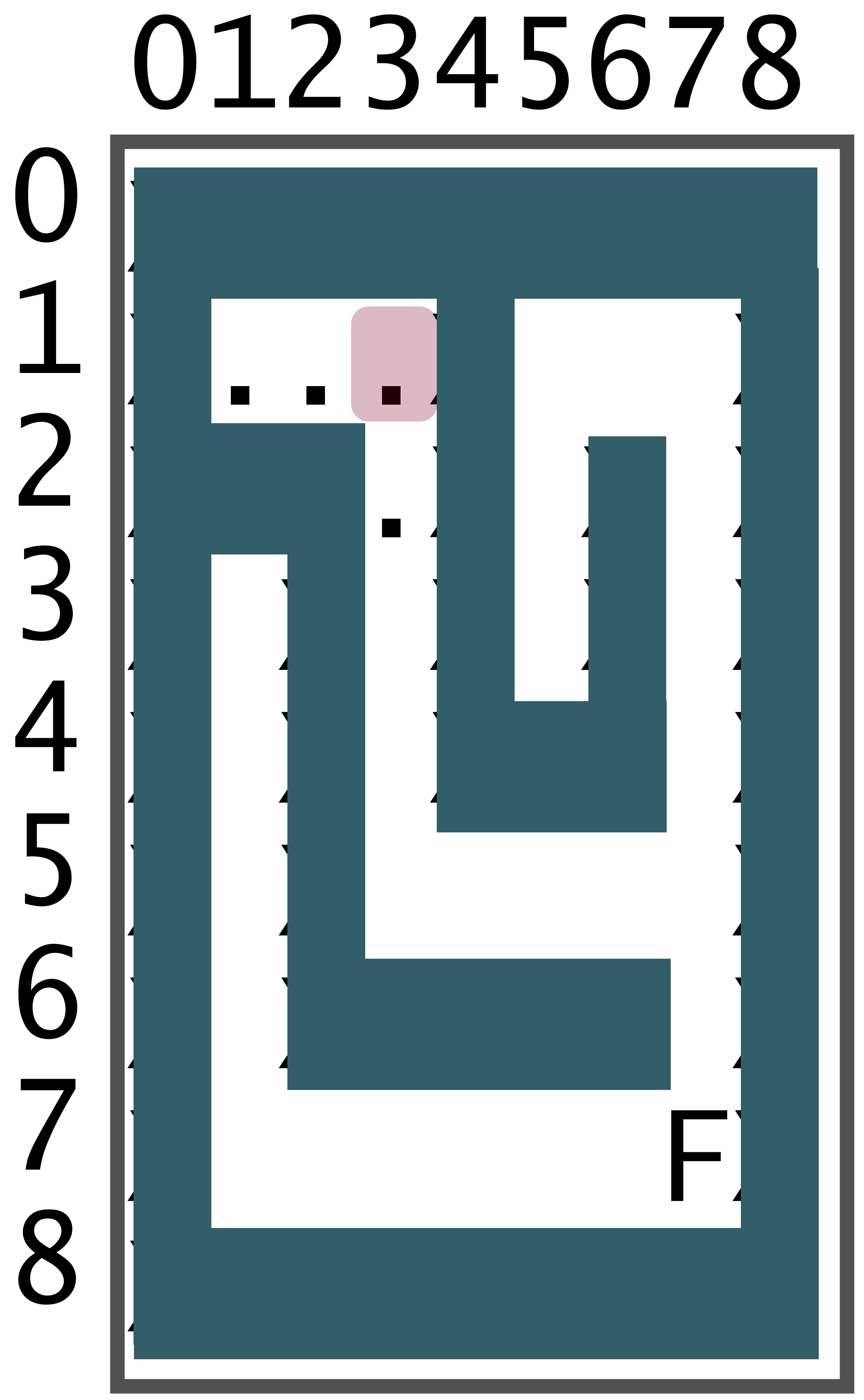
Maze Solving

- We will mark positions we have seen with a period ('.'), and mark backtracking with 'b'.
- Start: row=1 and col=1, Marking with period (.)
- We have to try all paths, N/E/S/W, and if we hit a wall ('X'), we can't go that direction.
- Trying north, row=0 and col=1, Hit wall! Back at row=1 and col=1,
- Trying east, row=1 and col=2, Marking with period (.)
- Trying north, row=0 and col=2, Hit wall! Back at row=1 and col=2,
- Trying east, row=1 and col=3, Marking with period (.)
- **Trying north, row=0 and col=3, Hit wall! Back at row=1 and col=3,**
- **Trying east, row=1 and col=4, Hit wall! Back at row=1 and col=3,**
- **Trying south, row=2 and col=3, Marking with period (.)**



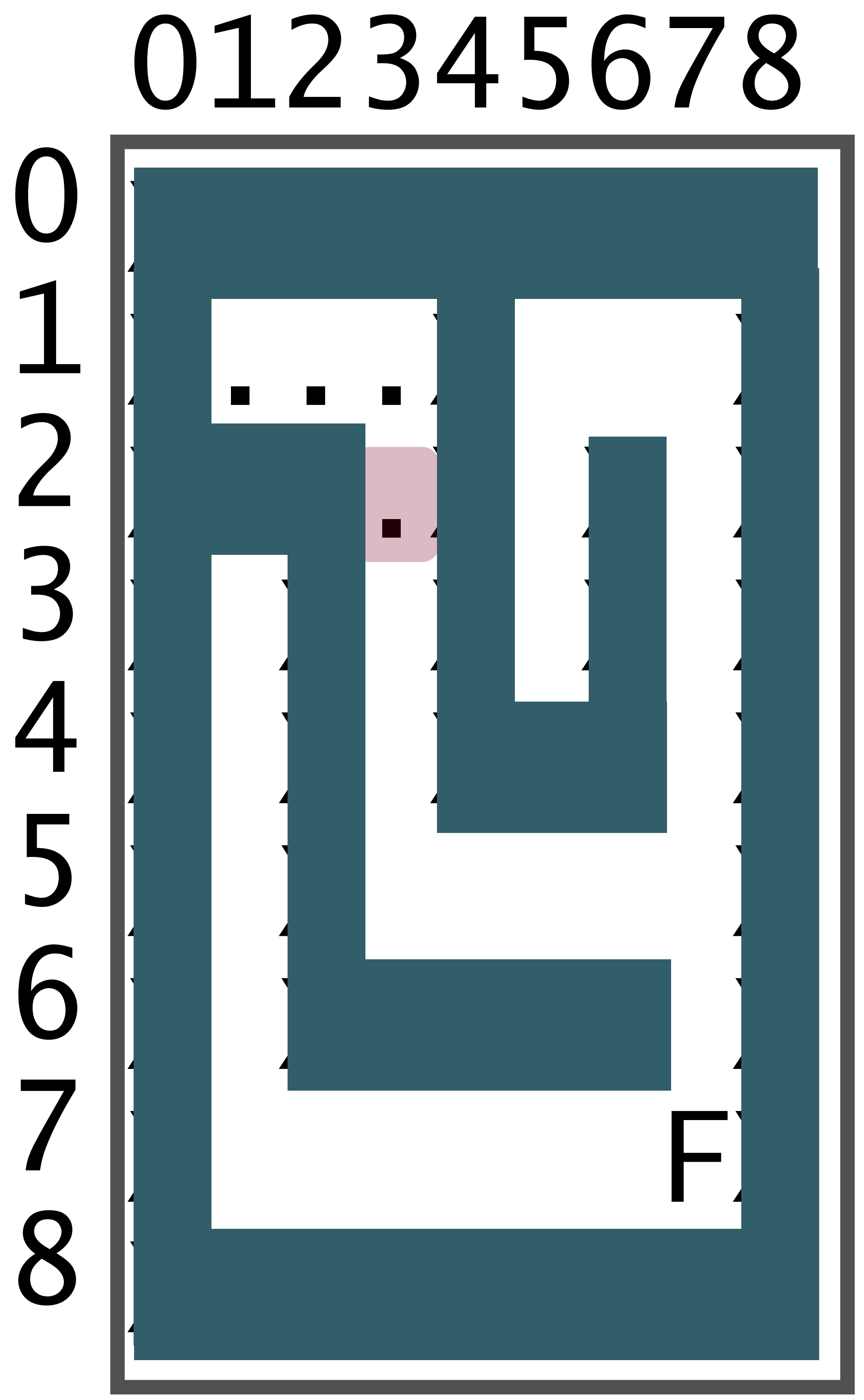
Maze Solving

- We will mark positions we have seen with a period ('.'), and mark backtracking with 'b'.
- Start: row=1 and col=1, Marking with period (.)
- We have to try all paths, N/E/S/W, and if we hit a wall ('X'), we can't go that direction.
- Trying north, row=0 and col=1, Hit wall! Back at row=1 and col=1,
- Trying east, row=1 and col=2, Marking with period (.)
- Trying north, row=0 and col=2, Hit wall! Back at row=1 and col=2,
- Trying east, row=1 and col=3, Marking with period (.)
- **Trying north, row=0 and col=3, Hit wall! Back at row=1 and col=3,**
- **Trying east, row=1 and col=4, Hit wall! Back at row=1 and col=3,**
- **Trying south, row=2 and col=3, Marking with period (.)**



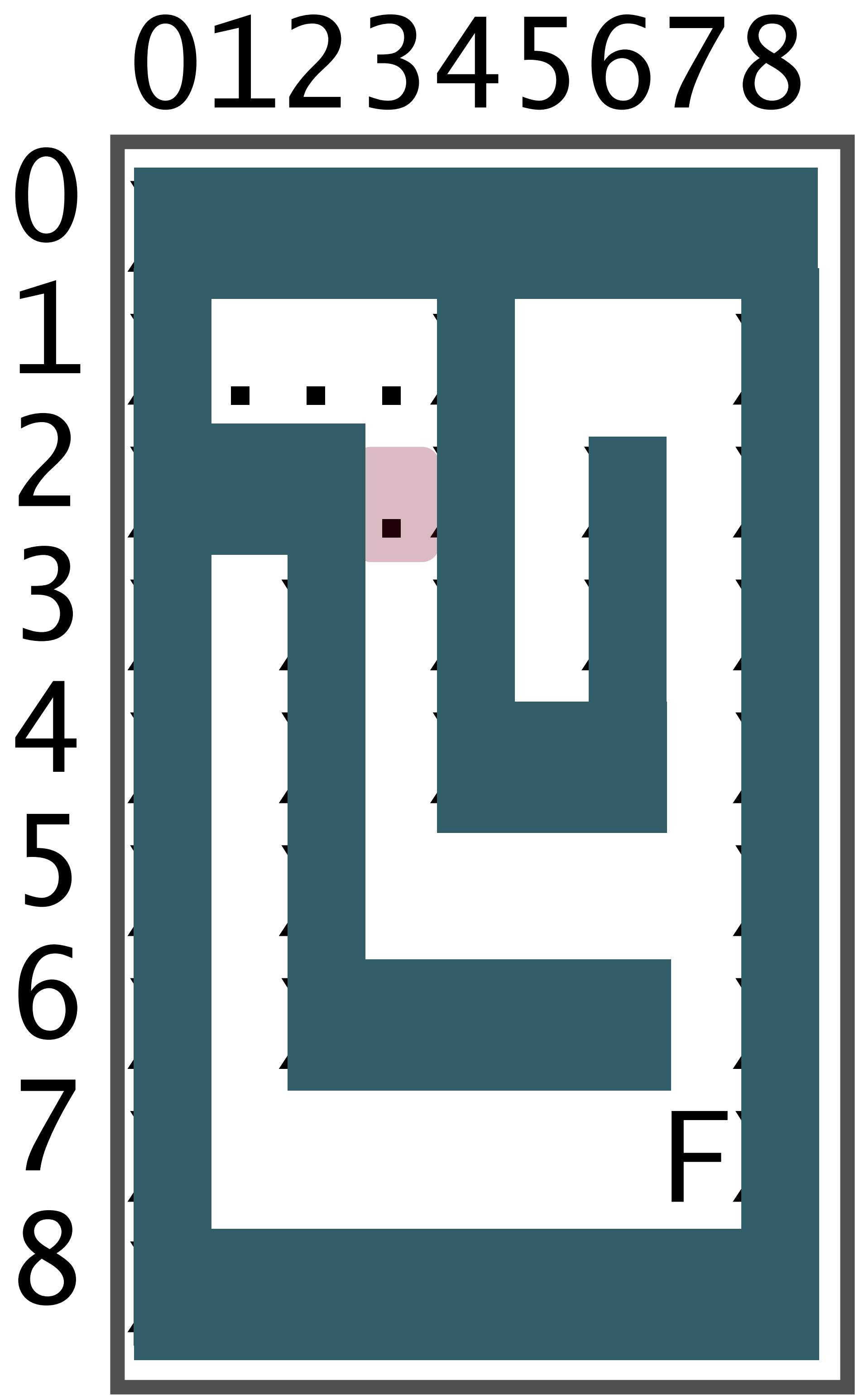
Maze Solving

- **Trying north, row=1 and col=3, We came from here! Back at row=2 and col=3,**



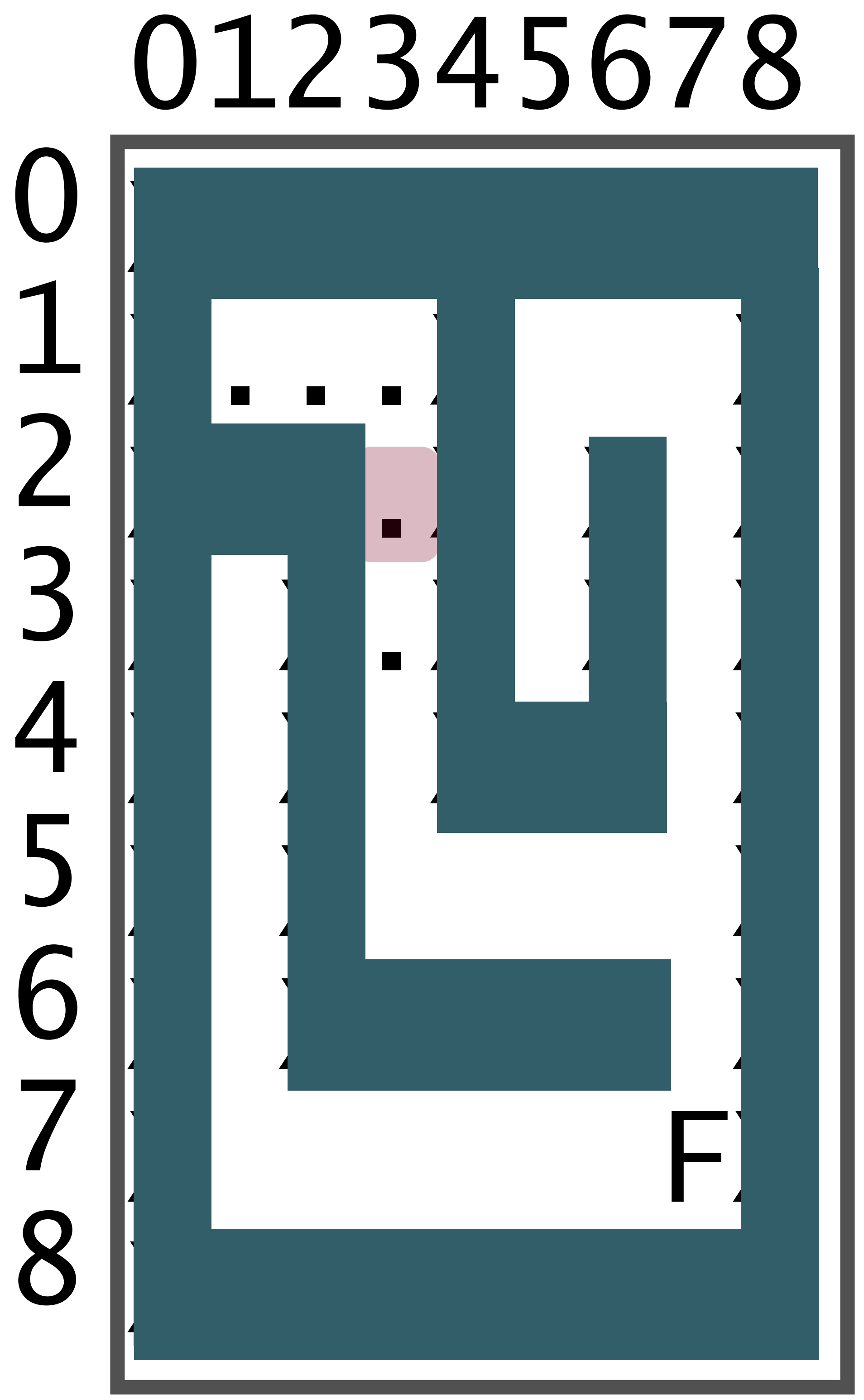
Maze Solving

- Trying north, row=1 and col=3, We came from here! Back at row=2 and col=3,
- **Trying east, row=2 and col=4, Hit wall! Back at row=2 and col=3,**
- **Trying south, row=3 and col=3, Marking with period (.)**



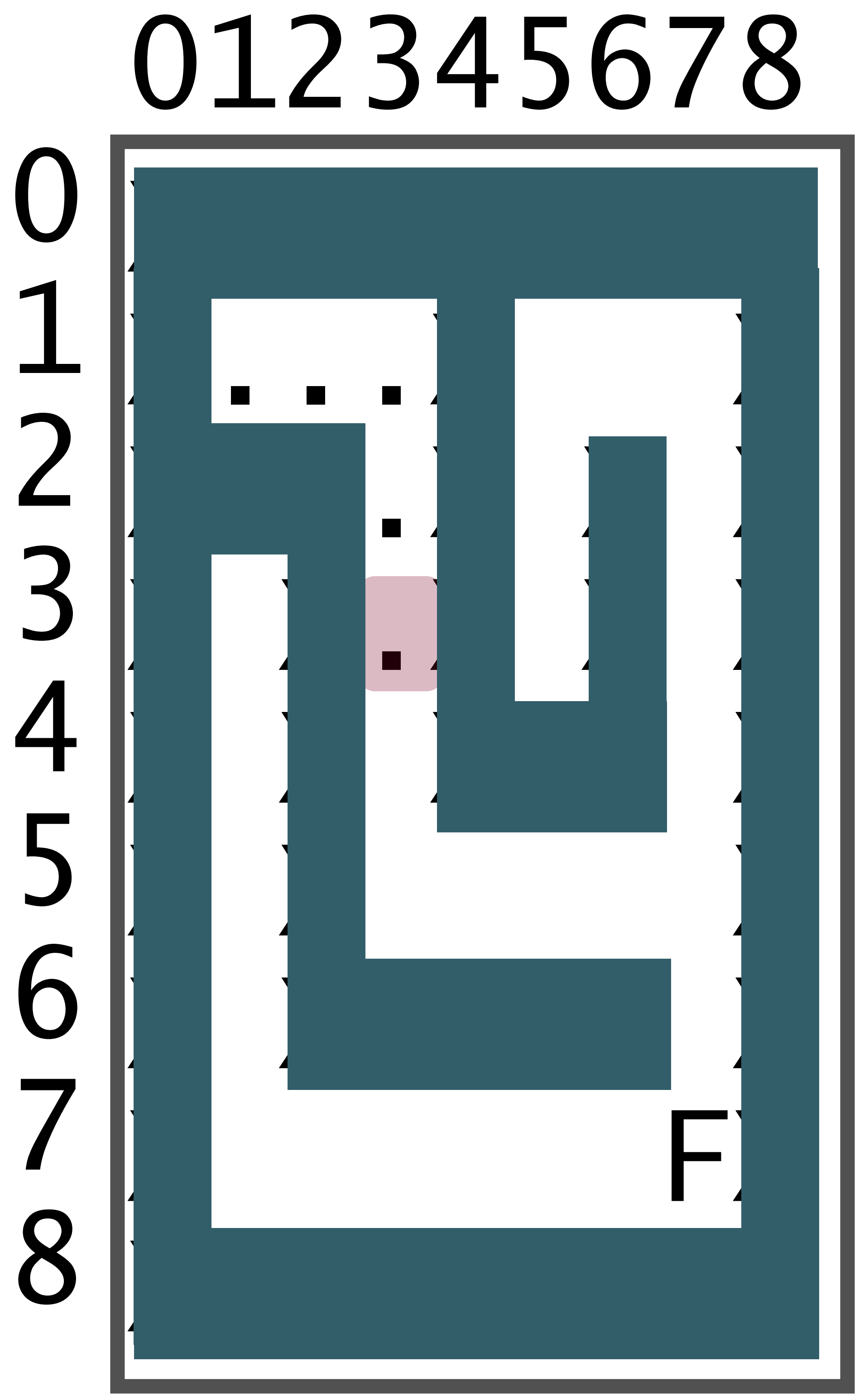
Maze Solving

- Trying north, row=1 and col=3, We came from here! Back at row=2 and col=3,
- **Trying east, row=2 and col=4, Hit wall! Back at row=2 and col=3,**
- **Trying south, row=3 and col=3, Marking with period (.)**



Maze Solving

- Trying north, row=1 and col=3, We came from here! Back at row=2 and col=3,
- Trying east, row=2 and col=4, Hit wall! Back at row=2 and col=3,
- Trying south, row=3 and col=3, Marking with period (.)
- **Trying north, row=2 and col=3, We came from here! Back at row=3 and col=3,**
- **Trying east, row=3 and col=4, Hit wall! Back at row=3 and col=3,**
- **Trying south, row=4 and col=3, Marking with period (.)**

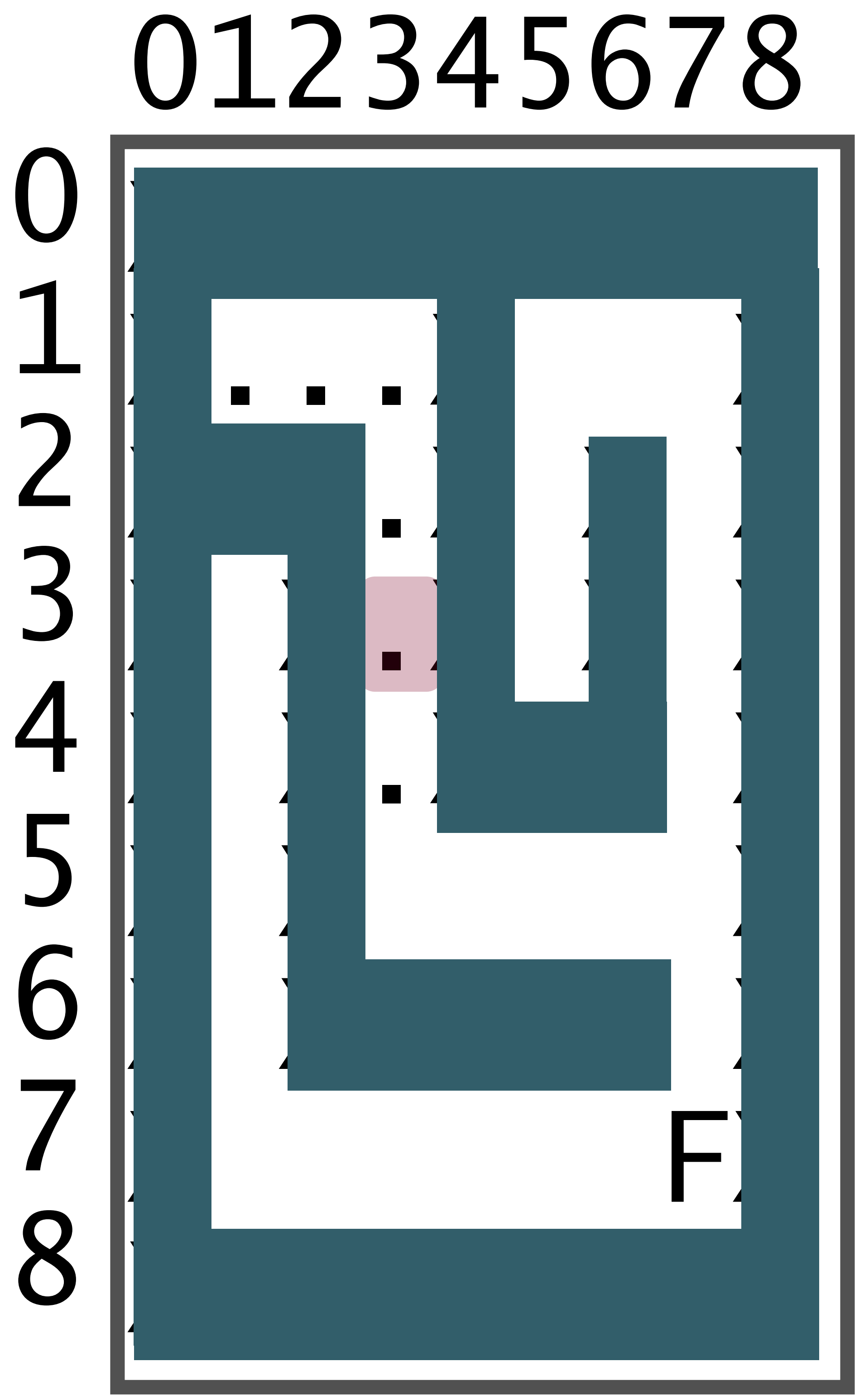


Maze Solving

- Trying north, row=1 and col=3, We came from here! Back at row=2 and col=3,
- Trying east, row=2 and col=4, Hit wall! Back at row=2 and col=3,
- Trying south, row=3 and col=3, Marking with period (.)
- **Trying north, row=2 and col=3, We came from here! Back at row=3 and col=3,**
- **Trying east, row=3 and col=4, Hit wall! Back at row=3 and col=3,**
- **Trying south, row=4 and col=3, Marking with period (.)**

...

(continues)

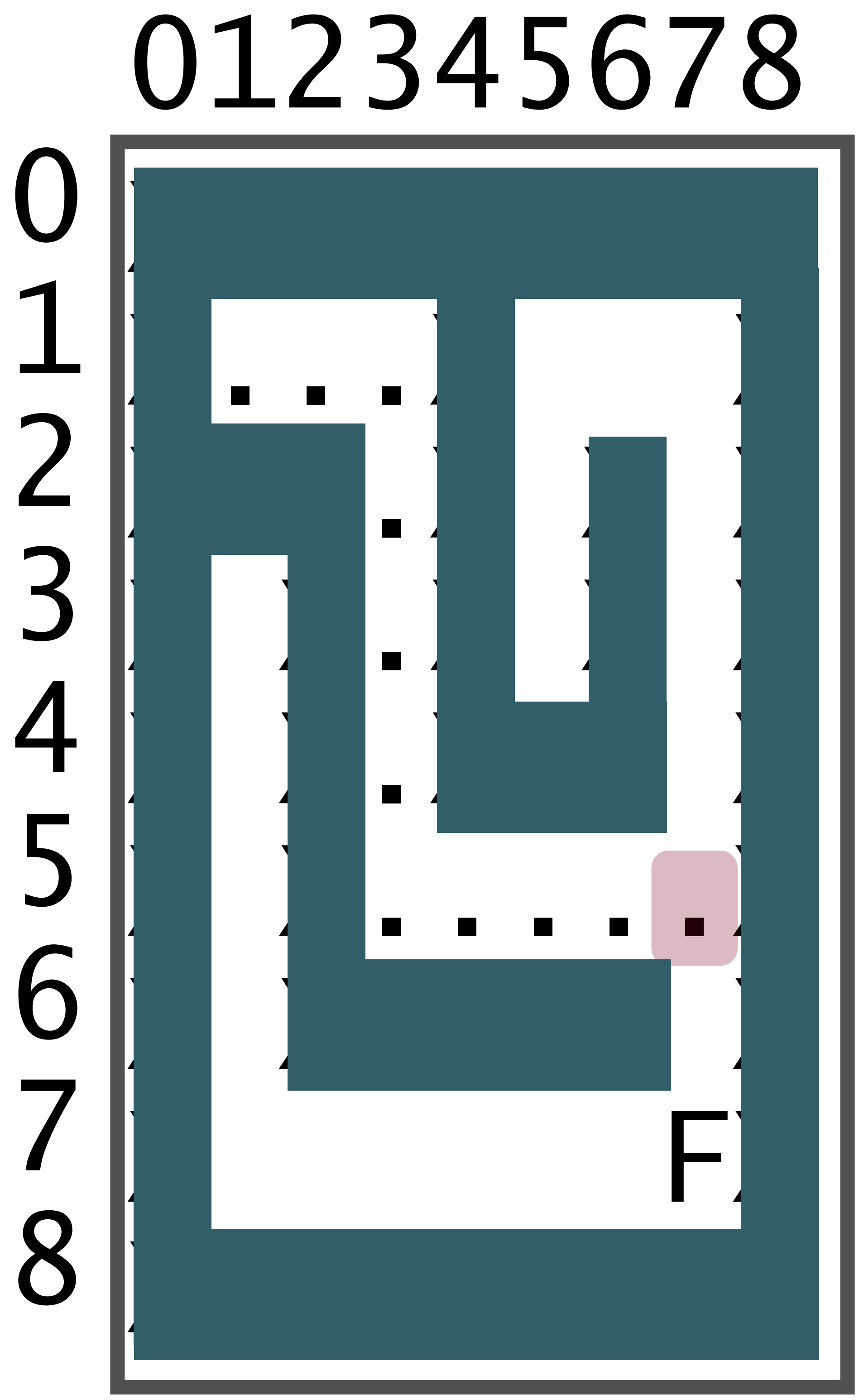


Maze Solving

- Trying north, row=1 and col=3, We came from here! Back at row=2 and col=3,
- Trying east, row=2 and col=4, Hit wall! Back at row=2 and col=3,
- Trying south, row=3 and col=3, Marking with period (.)
- Trying north, row=2 and col=3, We came from here! Back at row=3 and col=3,
- Trying east, row=3 and col=4, Hit wall! Back at row=3 and col=3,
- Trying south, row=4 and col=3, Marking with period (.)

...

(continues)



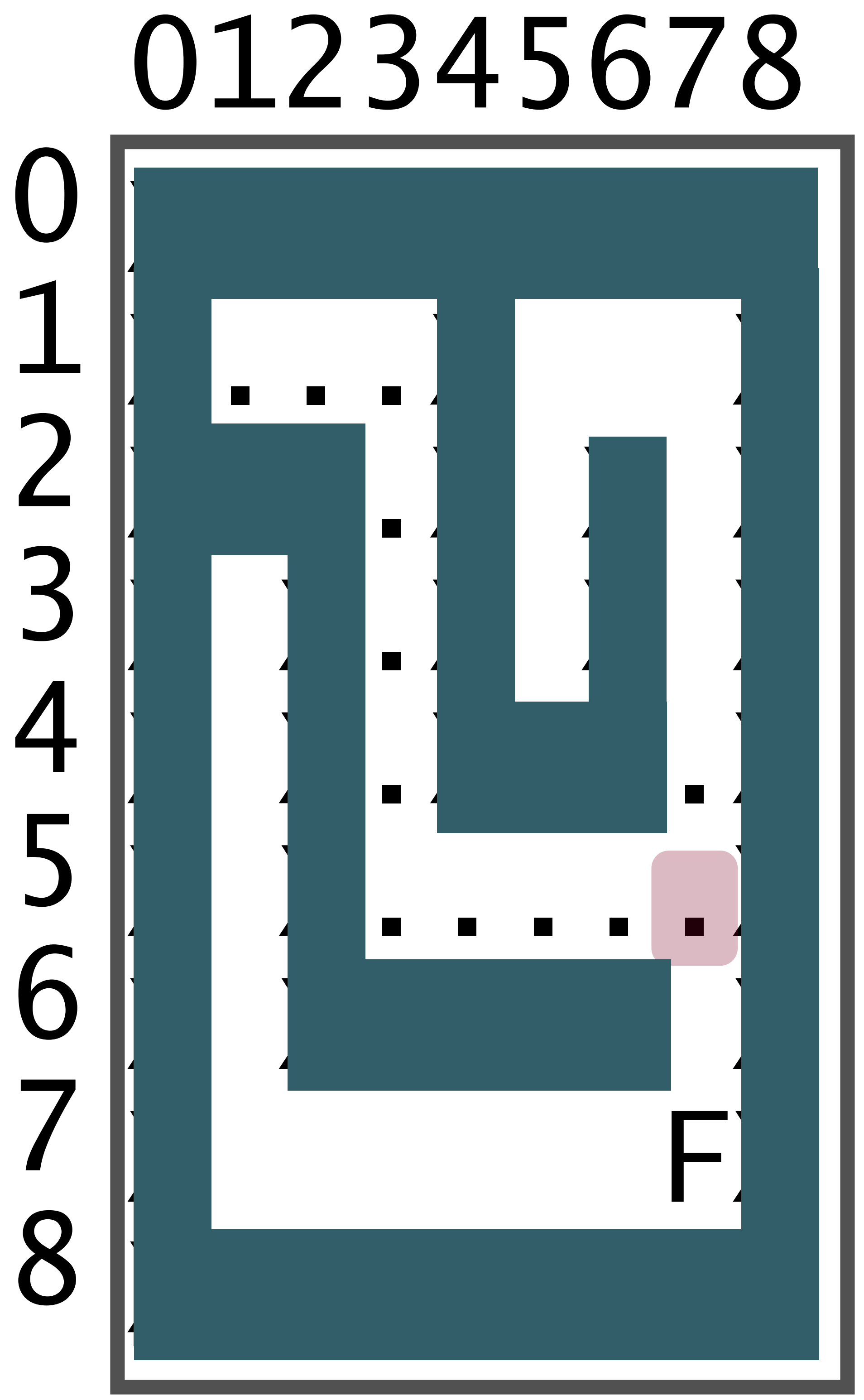
What happens here?

Maze Solving

- Trying north, row=1 and col=3, We came from here! Back at row=2 and col=3,
- Trying east, row=2 and col=4, Hit wall! Back at row=2 and col=3,
- Trying south, row=3 and col=3, Marking with period (.)
- Trying north, row=2 and col=3, We came from here! Back at row=3 and col=3,
- Trying east, row=3 and col=4, Hit wall! Back at row=3 and col=3,
- Trying south, row=4 and col=3, Marking with period (.)

...

(continues)

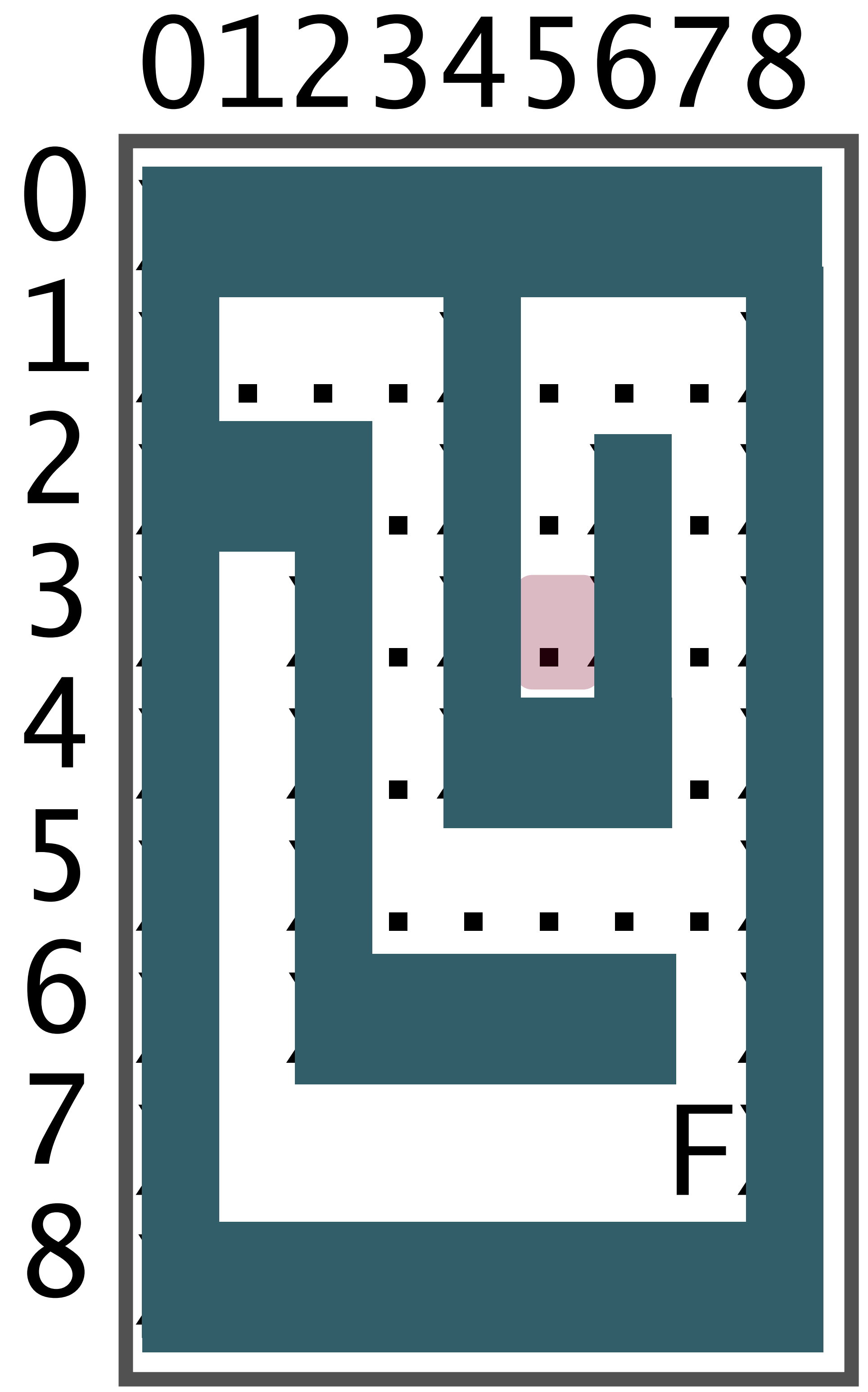


What happens here?

Bummer. We check North first, so we start going up.

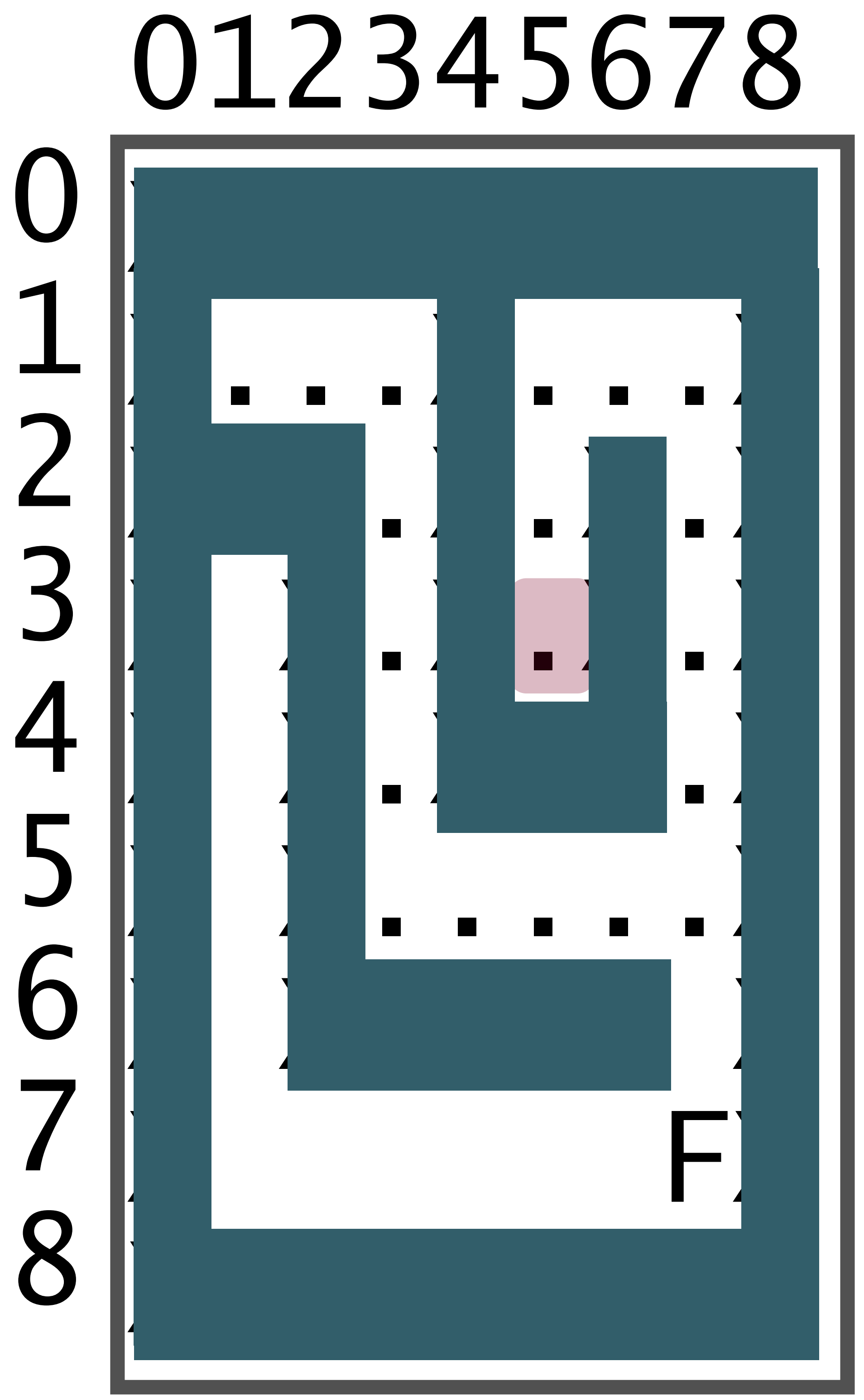
Maze Solving

Now what?



Maze Solving

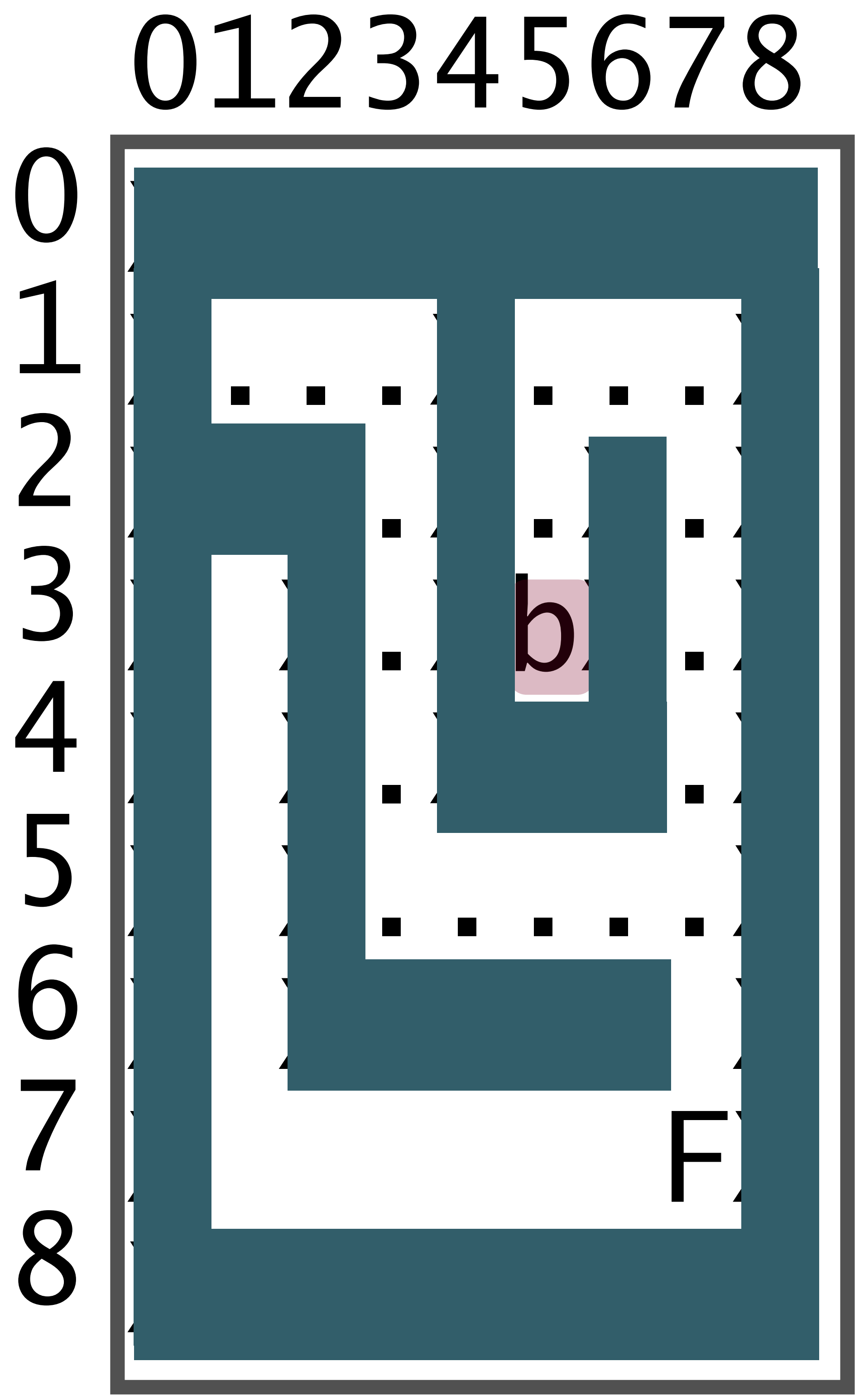
- Trying north, row=2 and col=5, We came from here! Back at row=3 and col=5,
- Trying east, row=3 and col=6, Hit wall! Back at row=3 and col=5,
- Trying south, row=4 and col=5, Hit wall! Back at row=3 and col=5,
- Trying west, row=3 and col=4, Hit wall! Back at row=3 and col=5,
- Failed. Marking bad path with b. Back at row=2 and col=5,



Maze Solving

- **Trying north, row=2 and col=5, We came from here! Back at row=3 and col=5,**
- **Trying east, row=3 and col=6, Hit wall! Back at row=3 and col=5,**
- **Trying south, row=4 and col=5, Hit wall! Back at row=3 and col=5,**
- **Trying west, row=3 and col=4, Hit wall! Back at row=3 and col=5,**
- **Failed. Marking bad path with b. Back at row=2 and col=5,**

What is next?
How did we get here? From the North, meaning we **checked South to get here.**
So, **we now check West (remember, we are checking N/E/S/W)**

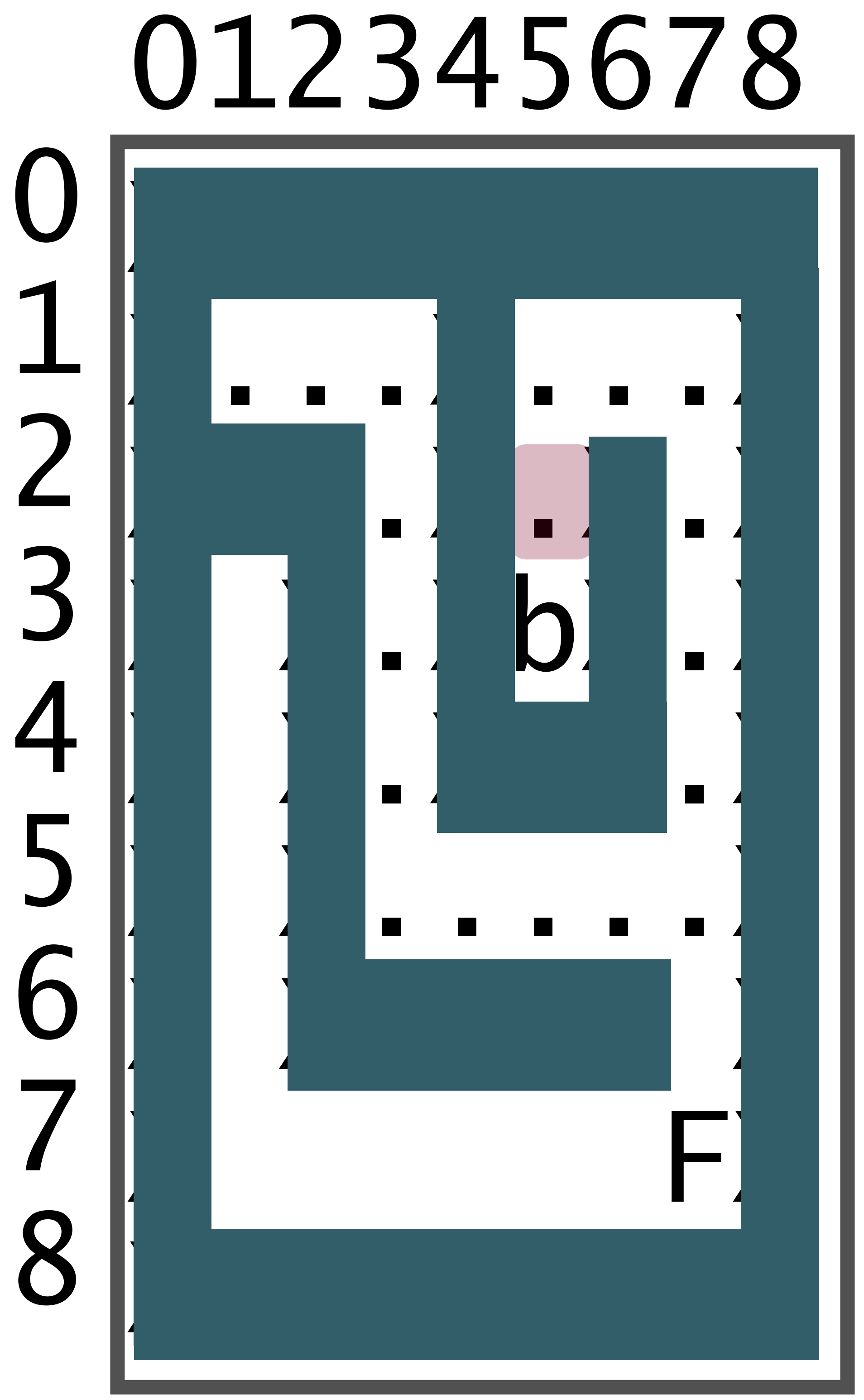


Maze Solving

- **Trying north, row=2 and col=5, We came from here! Back at row=3 and col=5,**
- **Trying east, row=3 and col=6, Hit wall! Back at row=3 and col=5,**
- **Trying south, row=4 and col=5, Hit wall! Back at row=3 and col=5,**
- **Trying west, row=3 and col=4, Hit wall! Back at row=3 and col=5,**
- **Failed. Marking bad path with b. Back at row=2 and col=5,**

What is next?
How did we get here? From the North, meaning we checked South to get here.
So, we now check West (remember, we are checking N/E/S/W)

Trying west, row=2 and col=4, Hit wall! Back at row=2 and col=5,
Failed. Marking bad path with b. Back at row=1 and col=5,

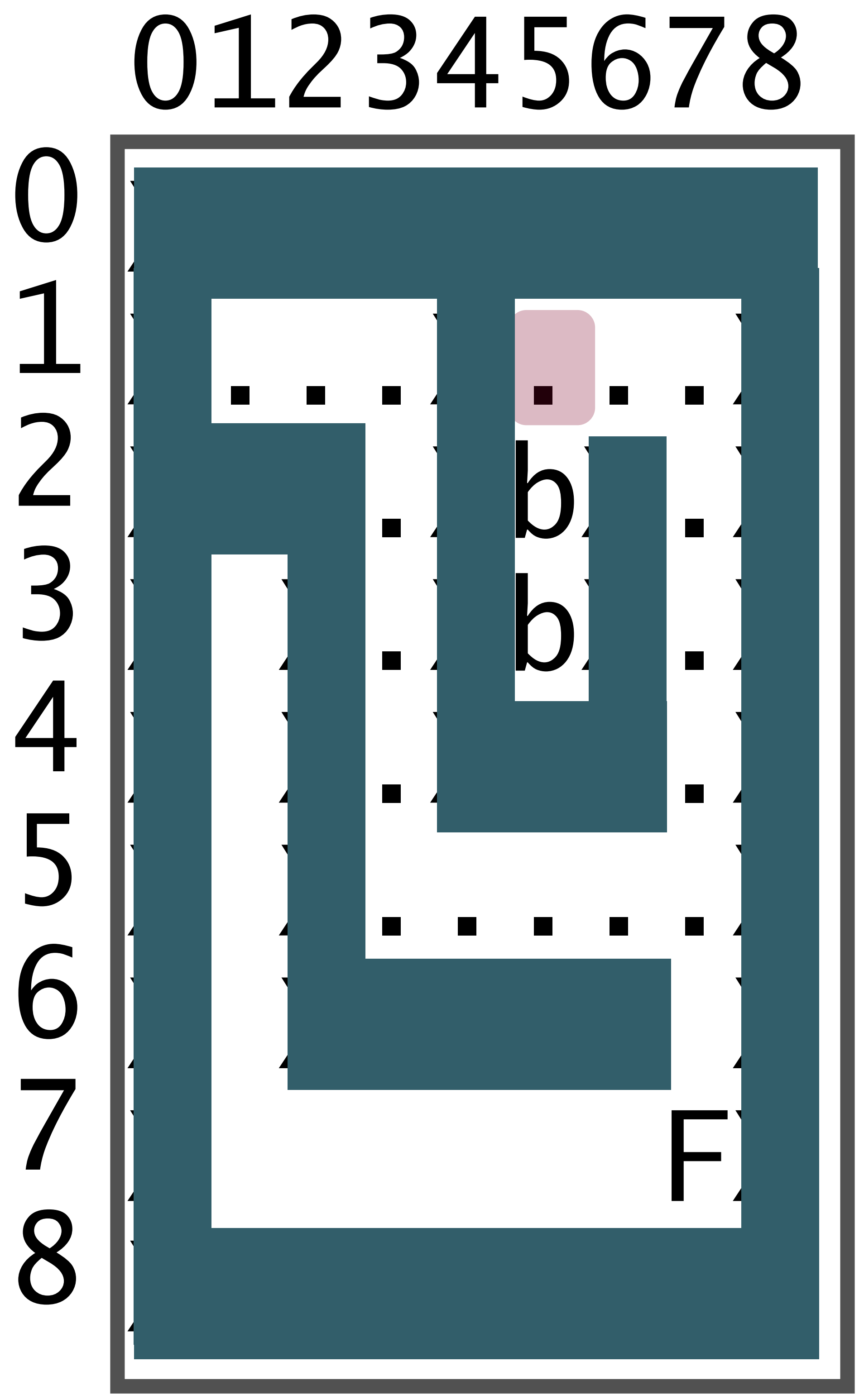


Maze Solving

- **Trying north, row=2 and col=5, We came from here! Back at row=3 and col=5,**
- **Trying east, row=3 and col=6, Hit wall! Back at row=3 and col=5,**
- **Trying south, row=4 and col=5, Hit wall! Back at row=3 and col=5,**
- **Trying west, row=3 and col=4, Hit wall! Back at row=3 and col=5,**
- **Failed. Marking bad path with b. Back at row=2 and col=5,**

What is next?
How did we get here? From the North, meaning we checked South to get here.
So, we now check West (remember, we are checking N/E/S/W)

Trying west, row=2 and col=4, Hit wall! Back at row=2 and col=5,
Failed. Marking bad path with b. Back at row=1 and col=5,



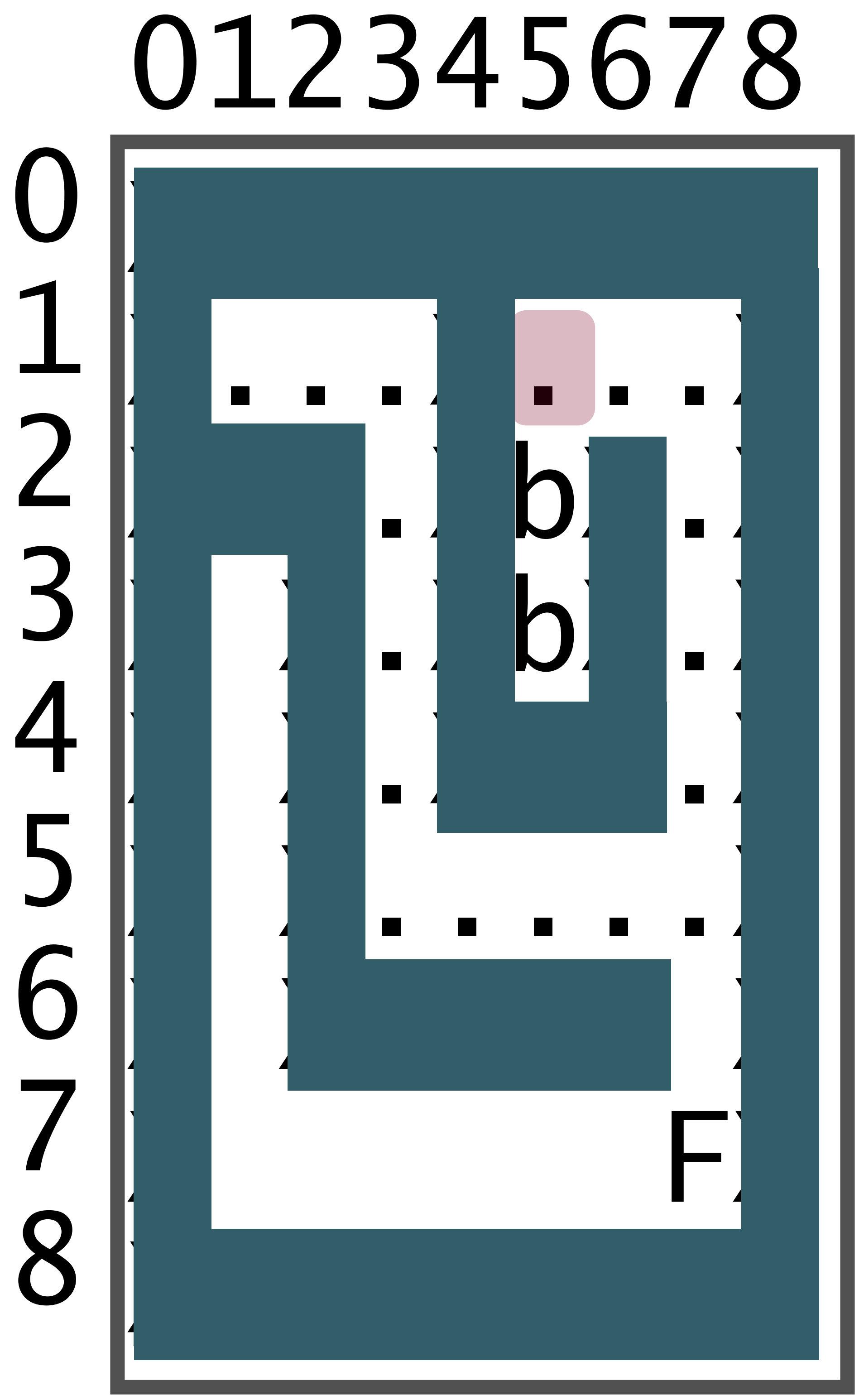
Maze Solving

- **Trying north, row=2 and col=5, We came from here! Back at row=3 and col=5,**
- **Trying east, row=3 and col=6, Hit wall! Back at row=3 and col=5,**
- **Trying south, row=4 and col=5, Hit wall! Back at row=3 and col=5,**
- **Trying west, row=3 and col=4, Hit wall! Back at row=3 and col=5,**
- **Failed. Marking bad path with b. Back at row=2 and col=5,**

What is next?
How did we get here? From the North, meaning we checked South to get here.
So, we now check West (remember, we are checking N/E/S/W)

Trying west, row=2 and col=4, Hit wall! Back at row=2 and col=5,
Failed. Marking bad path with b. Back at row=1 and col=5,

Now, we are "remembering" where we have been because we've been keeping track of our positions and what we last checked at a given position -- we will use recursion to do this!



Maze Solving

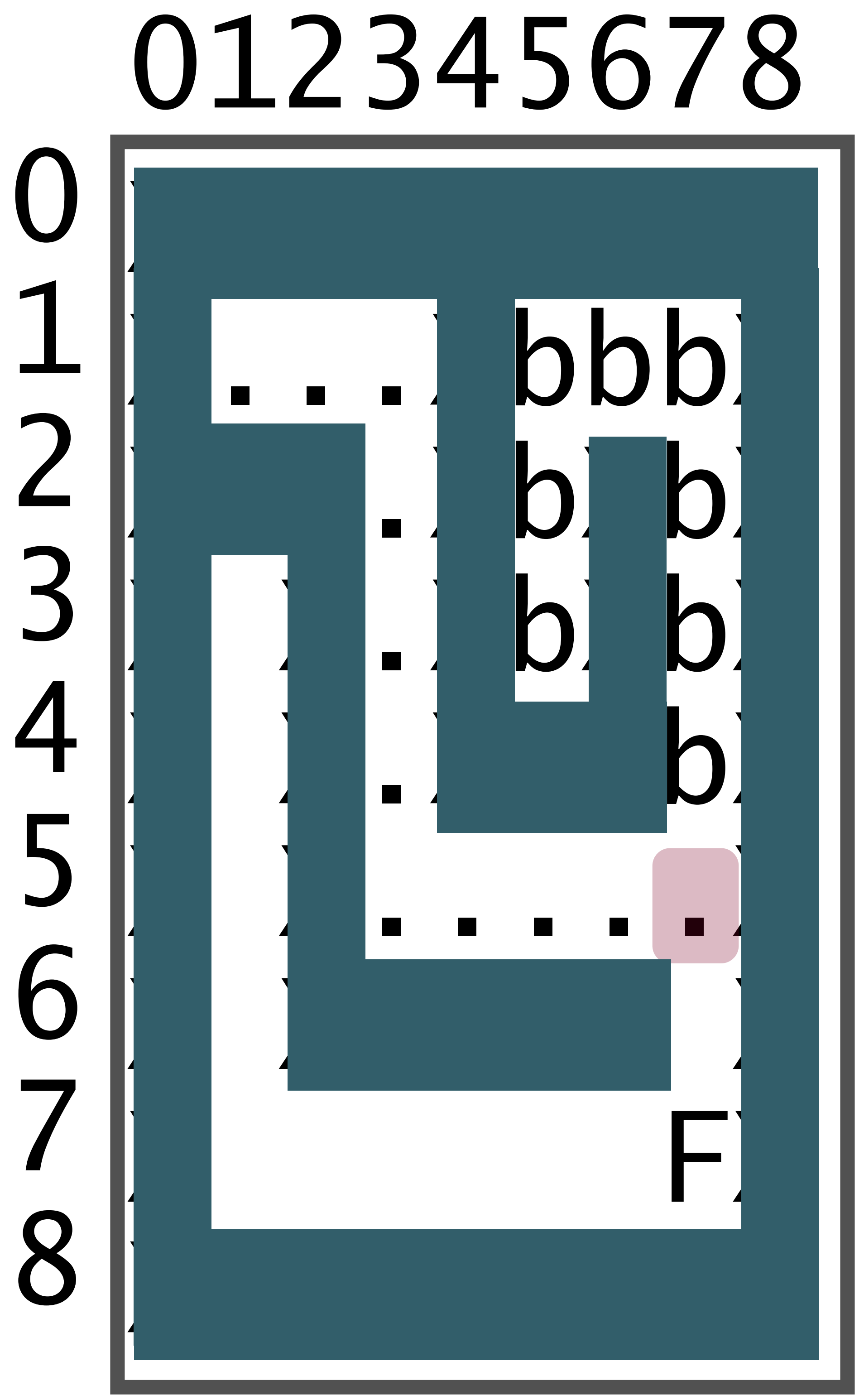
- **Trying north, row=2 and col=5, We came from here! Back at row=3 and col=5,**
- **Trying east, row=3 and col=6, Hit wall! Back at row=3 and col=5,**
- **Trying south, row=4 and col=5, Hit wall! Back at row=3 and col=5,**
- **Trying west, row=3 and col=4, Hit wall! Back at row=3 and col=5,**
- **Failed. Marking bad path with b. Back at row=2 and col=5,**

What is next?
How did we get here? From the North, meaning we checked South to get here.
So, we now check West (remember, we are checking N/E/S/W)

Trying west, row=2 and col=4, Hit wall! Back at row=2 and col=5,
Failed. Marking bad path with b. Back at row=1 and col=5,

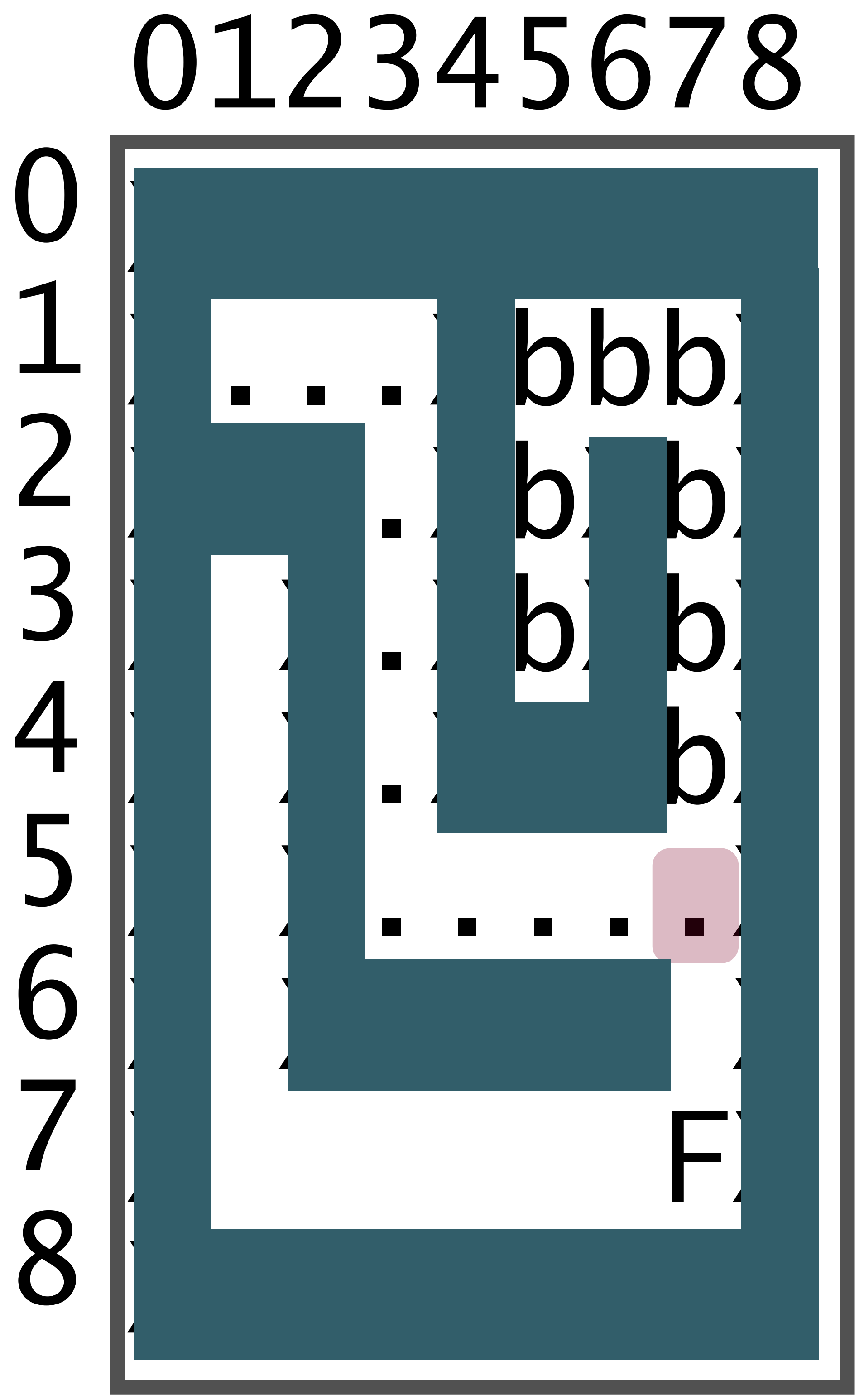
Now, we are "remembering" where we have been because we've been keeping track of our positions and what we last checked at a given position -- we will use recursion to do this!

We will arrive back at row=5, col=7 quickly.



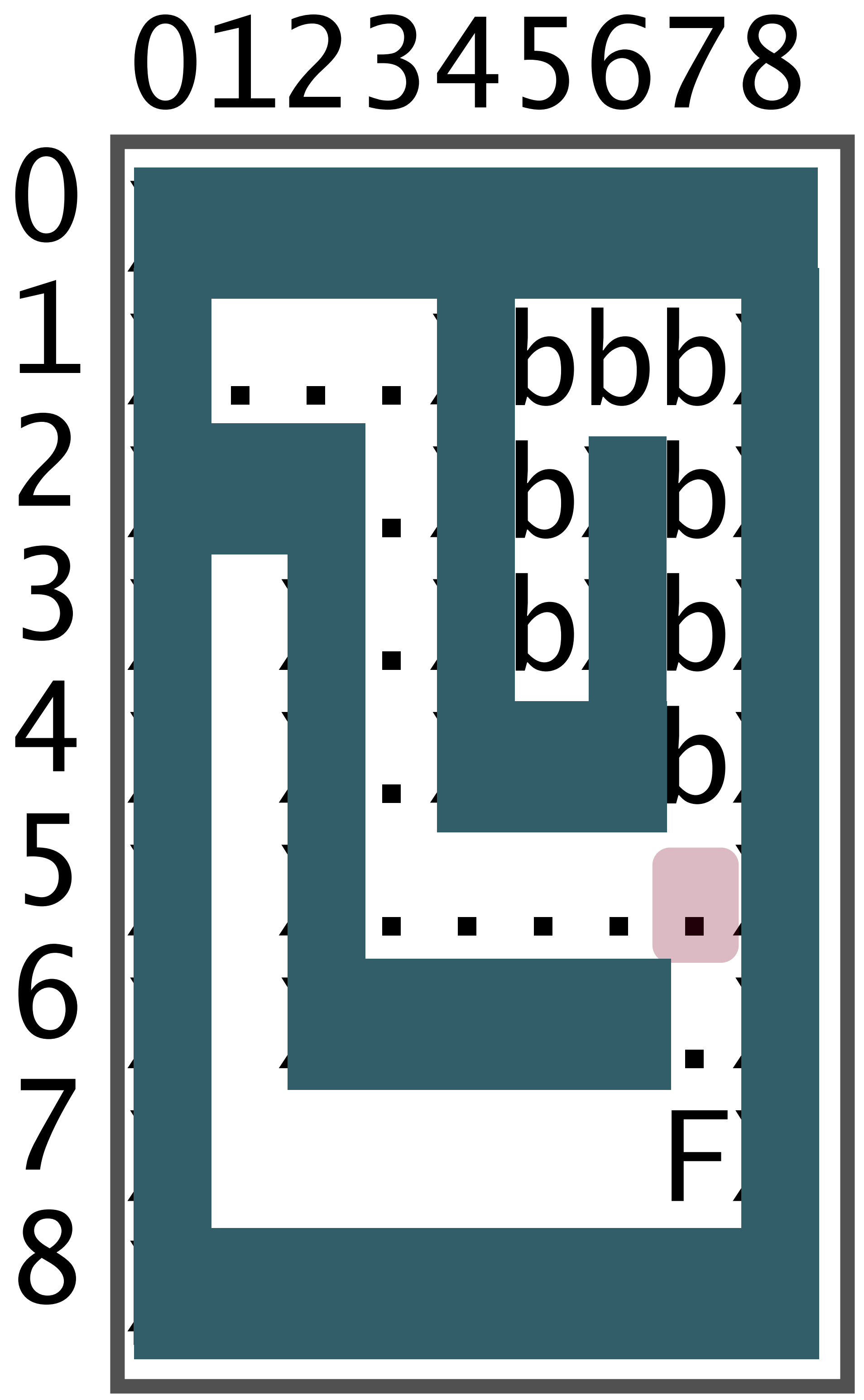
Maze Solving

- **Trying east, row=5 and col=8, Hit wall! Back at row=5 and col=7,**
- **Trying south, row=6 and col=7, Marking with period (.)**



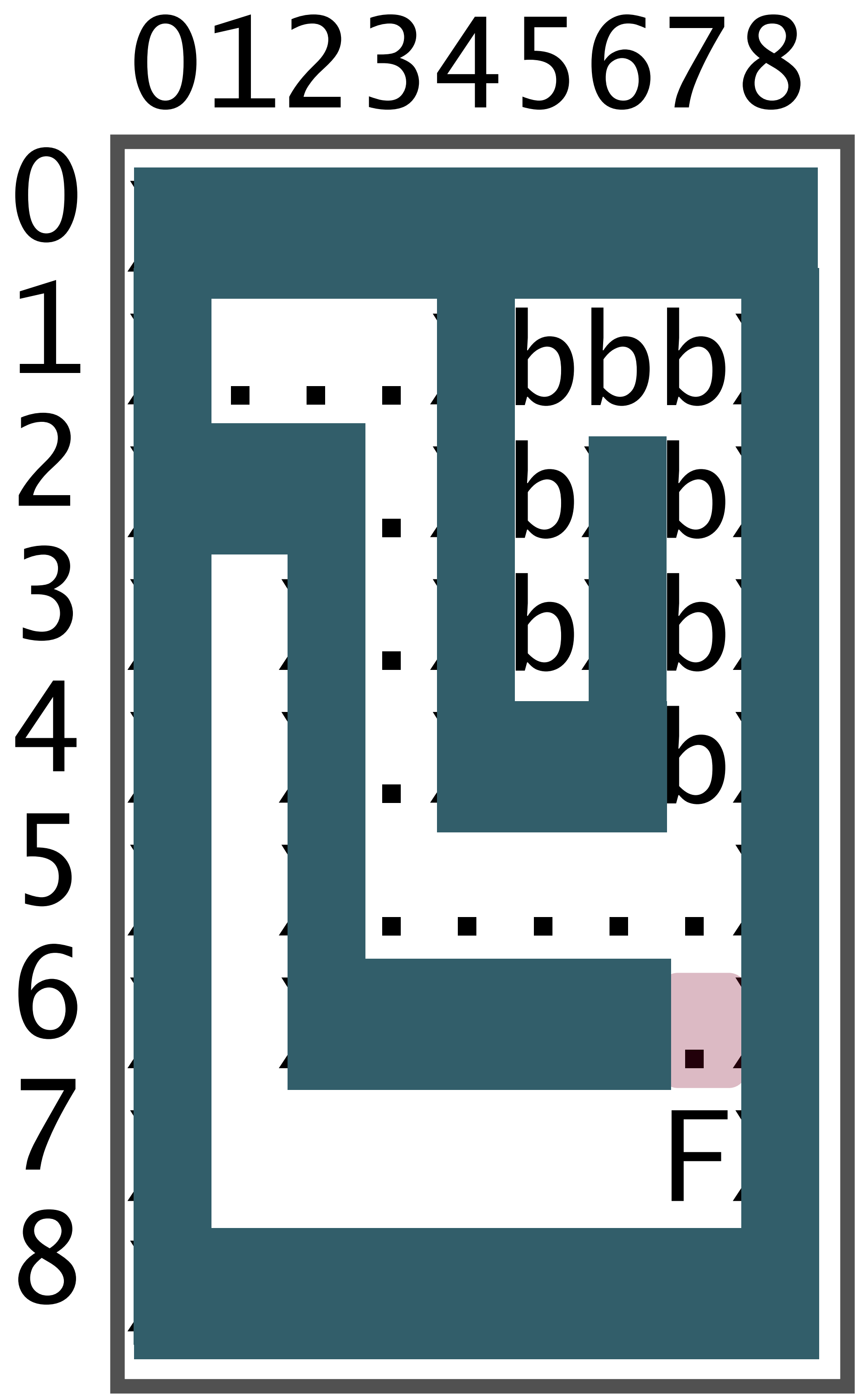
Maze Solving

- **Trying east, row=5 and col=8, Hit wall! Back at row=5 and col=7,**
- **Trying south, row=6 and col=7, Marking with period (.)**



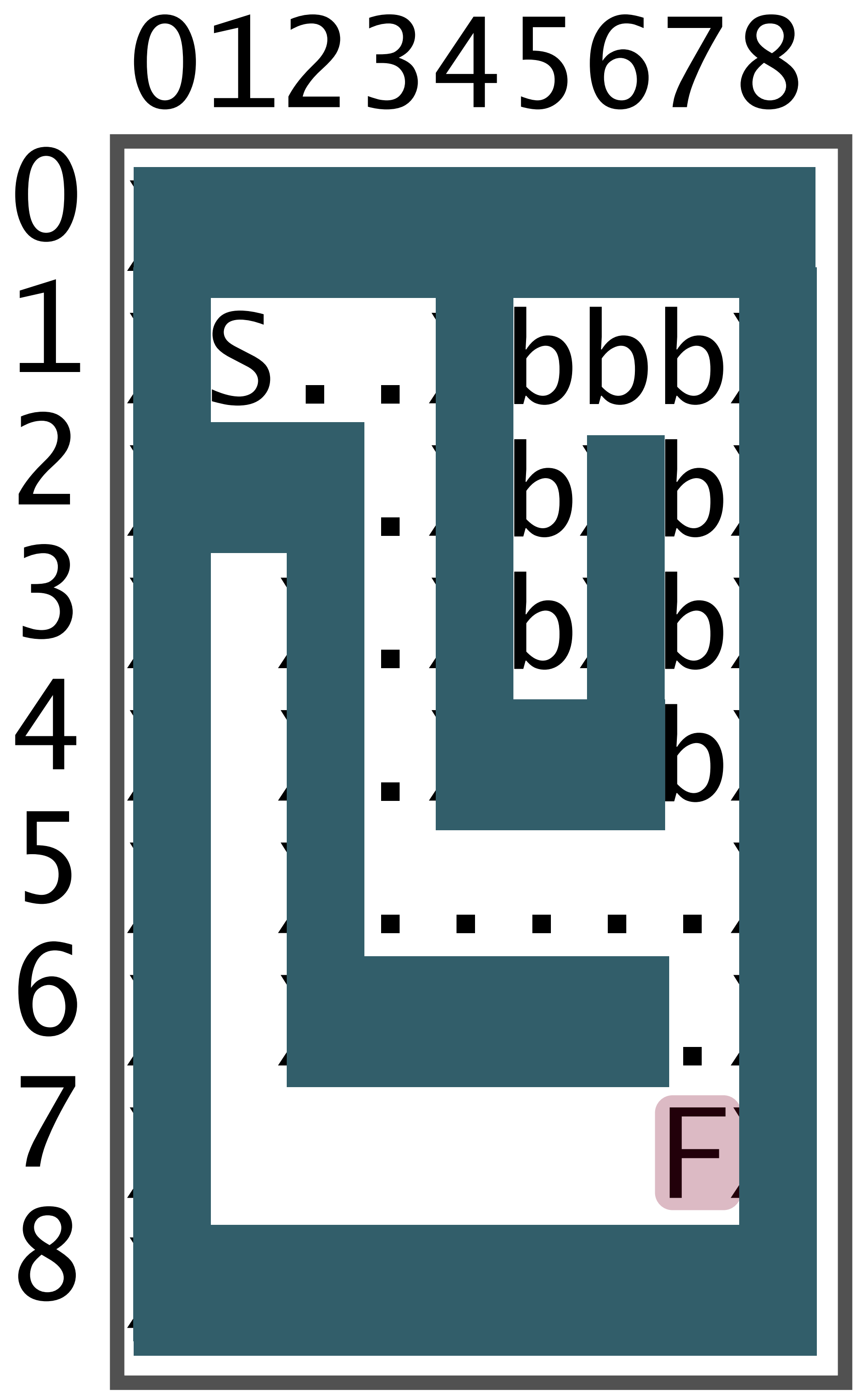
Maze Solving

- Trying east, row=5 and col=8, Hit wall! Back at row=5 and col=7,
- Trying south, row=6 and col=7, Marking with period (.)
- **Trying north, row=5 and col=7, We came from here! Back at row=6 and col=7,**
- **Trying east, row=6 and col=8, Hit wall! Back at row=6 and col=7,**
- **Trying south, row=7 and col=7, Found the Finish!**



Maze Solving

- Trying east, row=5 and col=8, Hit wall! Back at row=5 and col=7,
- Trying south, row=6 and col=7, Marking with period (.)
- **Trying north, row=5 and col=7, We came from here! Back at row=6 and col=7,**
- **Trying east, row=6 and col=8, Hit wall! Back at row=6 and col=7,**
- **Trying south, row=7 and col=7, Found the Finish!**



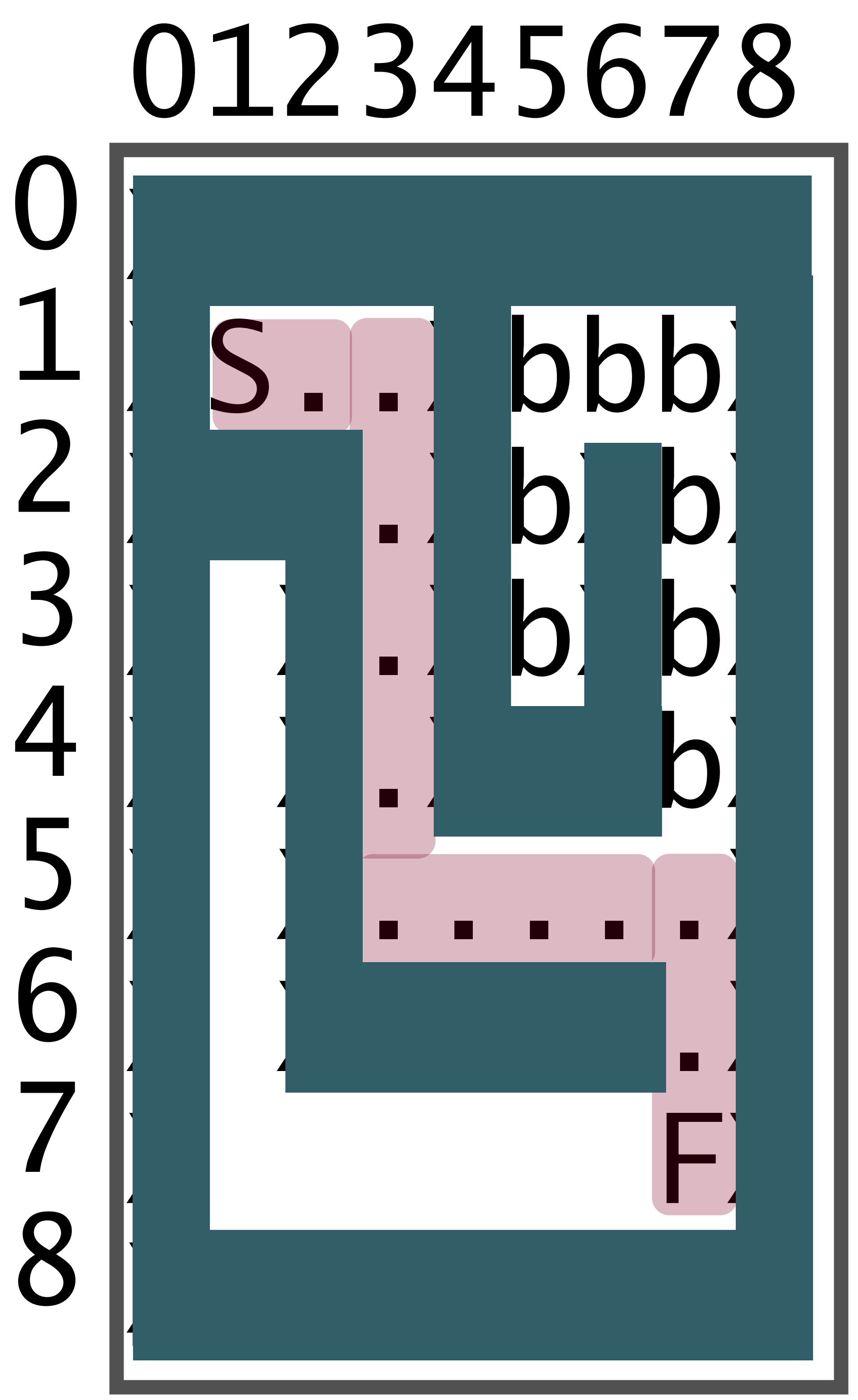
Maze Solving

- Trying east, row=5 and col=8, Hit wall! Back at row=5 and col=7,
- Trying south, row=6 and col=7, Marking with period (.)
- Trying north, row=5 and col=7, We came from here! Back at row=6 and col=7,
- Trying east, row=6 and col=8, Hit wall! Back at row=6 and col=7,
- Trying south, row=7 and col=7, Found the Finish!

The total number of steps: 71!

That seems like a lot of steps to solve such a small maze, but remember, we are going through a methodical process that *must check all paths*.

(see extra slides for all steps for this maze)



Maze Solving

- Our recursive backtracking method for solving mazes must follow the same rules for all recursion:

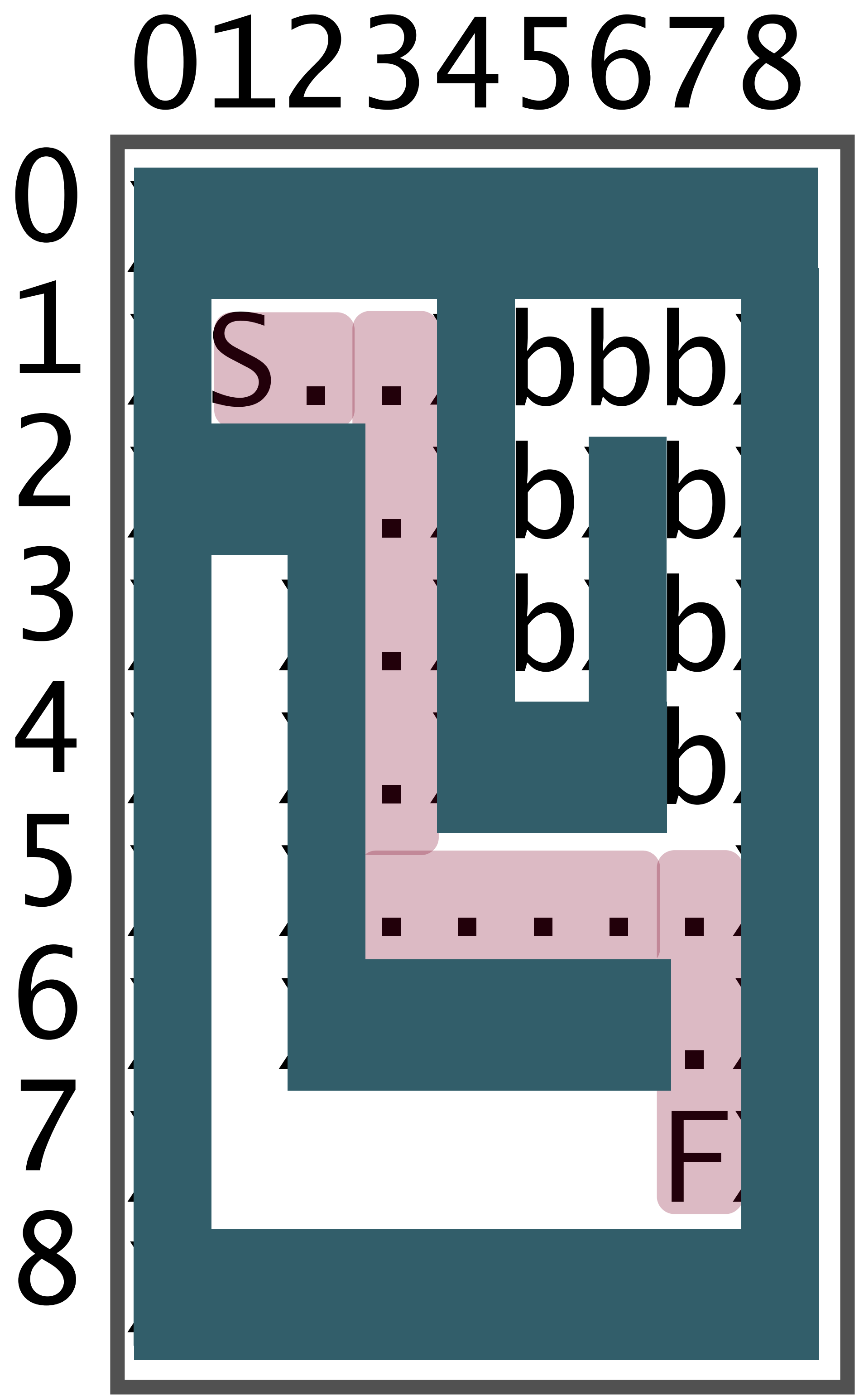
(1) have a case for all valid inputs,
(2) must have base cases,
(3) make forward progress towards the base case.

Let's start with the base cases. How many are there?

(1) If we go out of the bounds of the maze (the grid bounds).
- This actually won't happen for our mazes, because we have surrounded all paths with walls.

(2) If we hit a backtracked position ('b')
- Also won't happen, because once we mark as backtracked, we'll never get there again.

(3) If we hit a wall ('X')
(4) If we hit a position we have seen before ('.')
(5) If we find the finish ('F')



Maze Solving

Base cases:

Returning **`true`** means we have solved the maze!

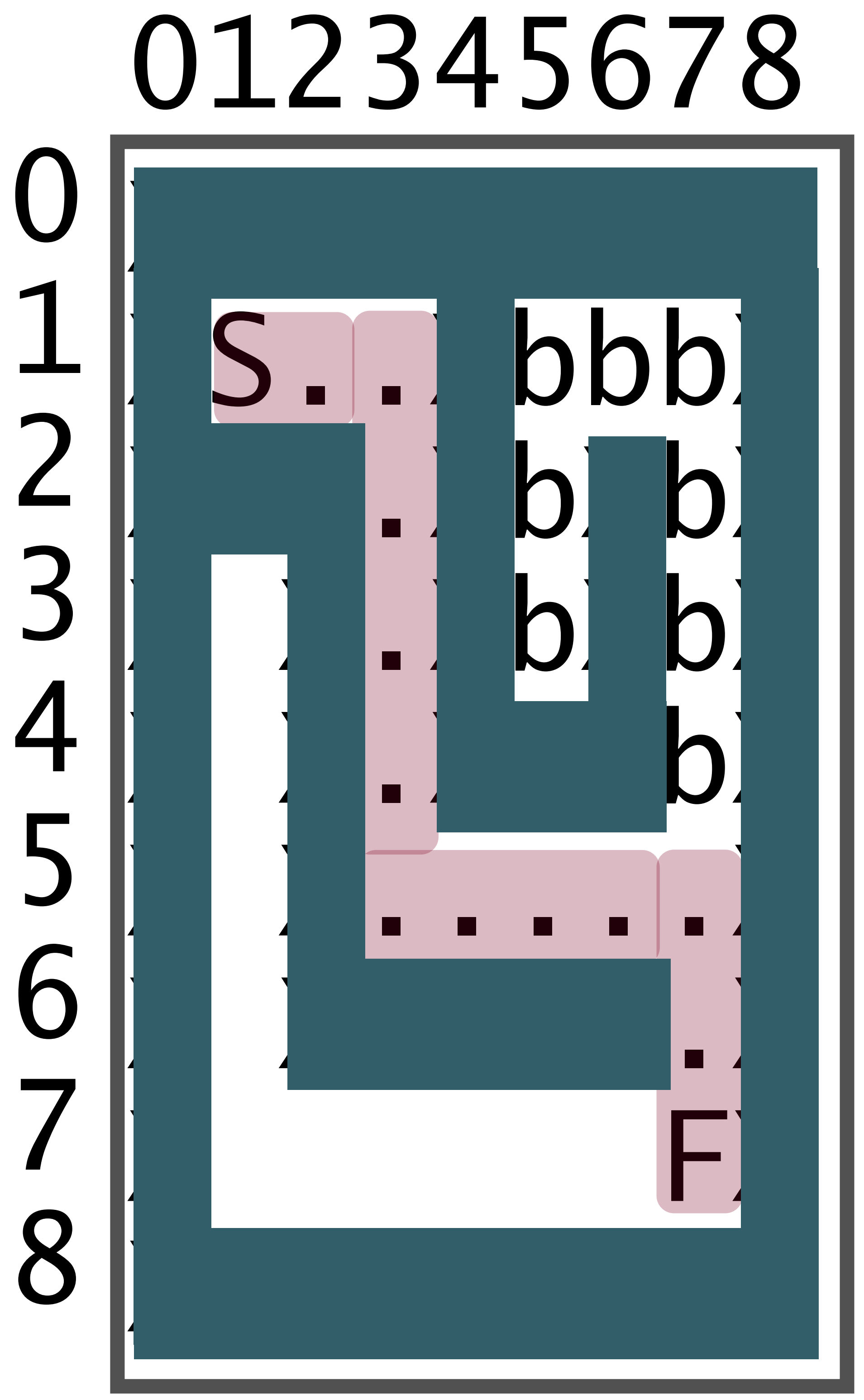
Returning **`false`** means that this path does not solve the maze.

```
bool solveMazeRecursive(int row, int col, Grid<int> &maze) {

    if (maze[row][col] == 'X') {
        return false;
    }

    if (maze[row][col] == '.') {
        return false;
    }

    if (maze[row][col] == 'F') {
        return true;
    }
}
```



Maze Solving

Once we take care of our base cases, we'd better mark the position we are at!

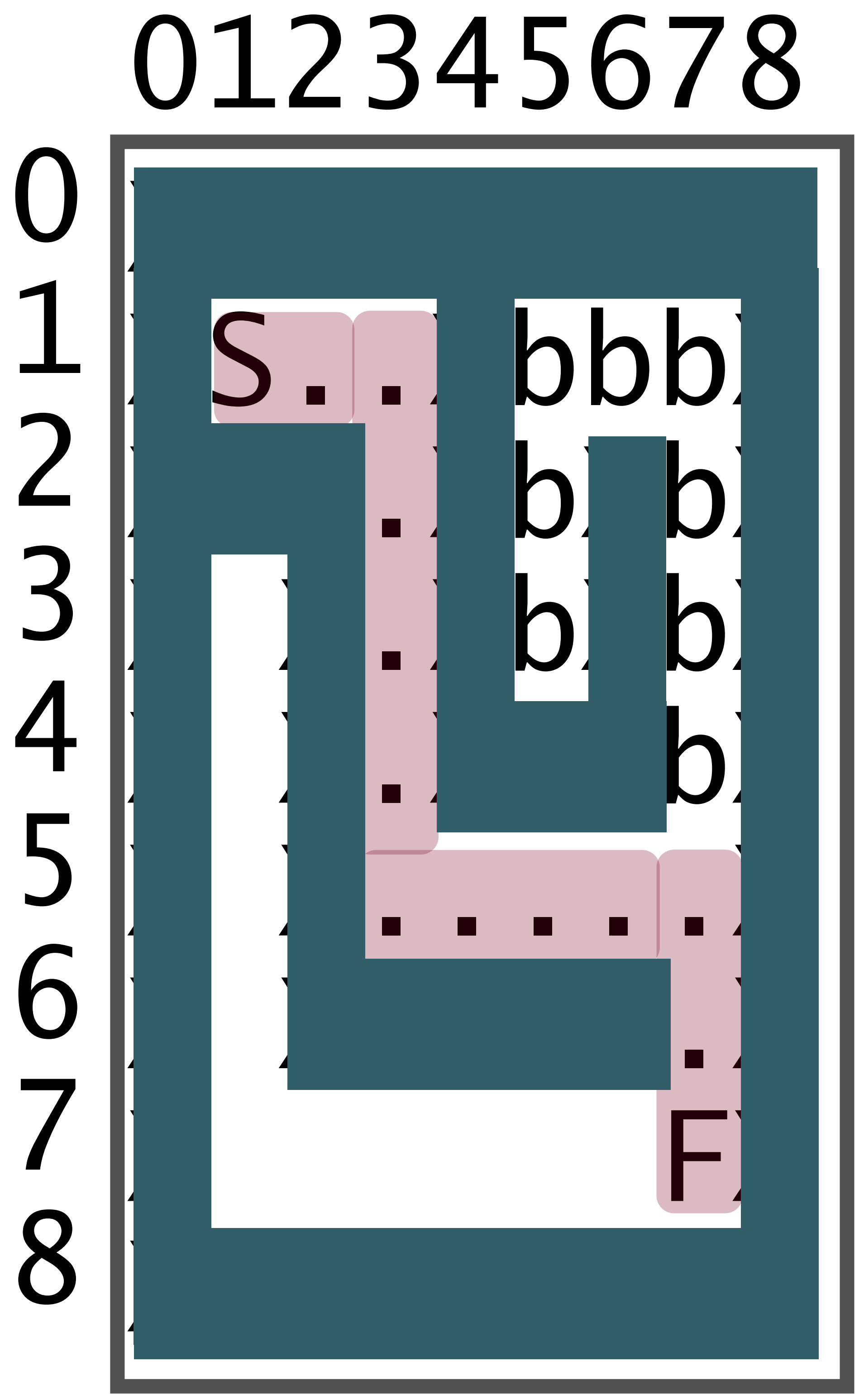
```
bool solveMazeRecursive(int row, int col, Grid<int> &maze) {

    if (maze[row][col] == 'X') {
        return false;
    }

    if (maze[row][col] == '.') {
        return false;
    }

    if (maze[row][col] == 'F') {
        return true;
    }

    maze[row][col] = '.';
}
```



Maze Solving

Now we can recurse -- we have to check all directions!

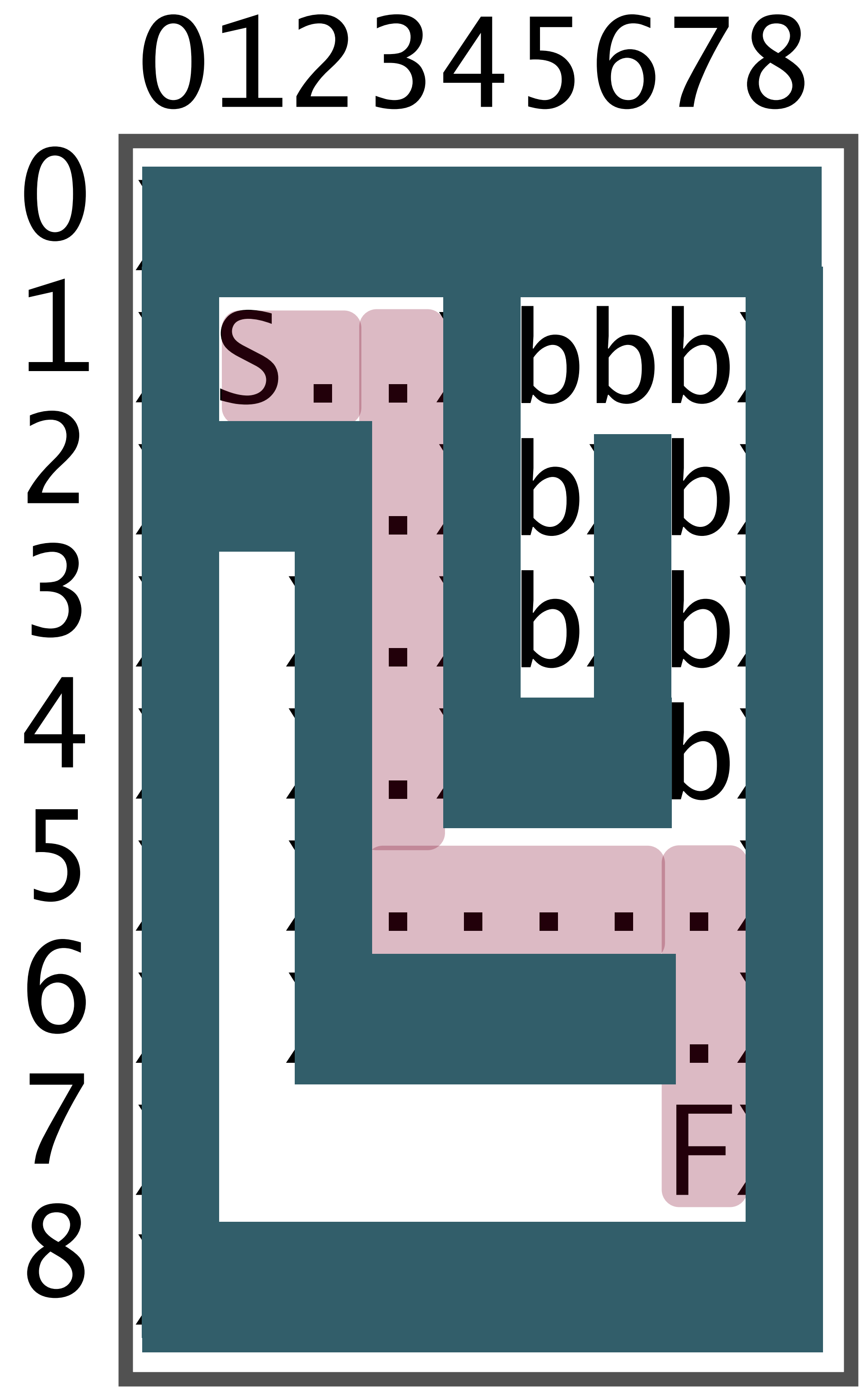
```
bool solveMazeRecursive(int row, int col, Grid<int> &maze) {

    ...

    maze[row][col] = '.';

    // Recursively call solveMazeRecursivePrint(row,col)
    // for north, east, south, and west
    // If one of the positions returns true, then return true

    // north
    if (solveMazeRecursivePrint(row-1,col,maze)) {
        return true;
    }
    ...
}
```



Maze Solving

All four recursions. If all four return, we have to backtrack!

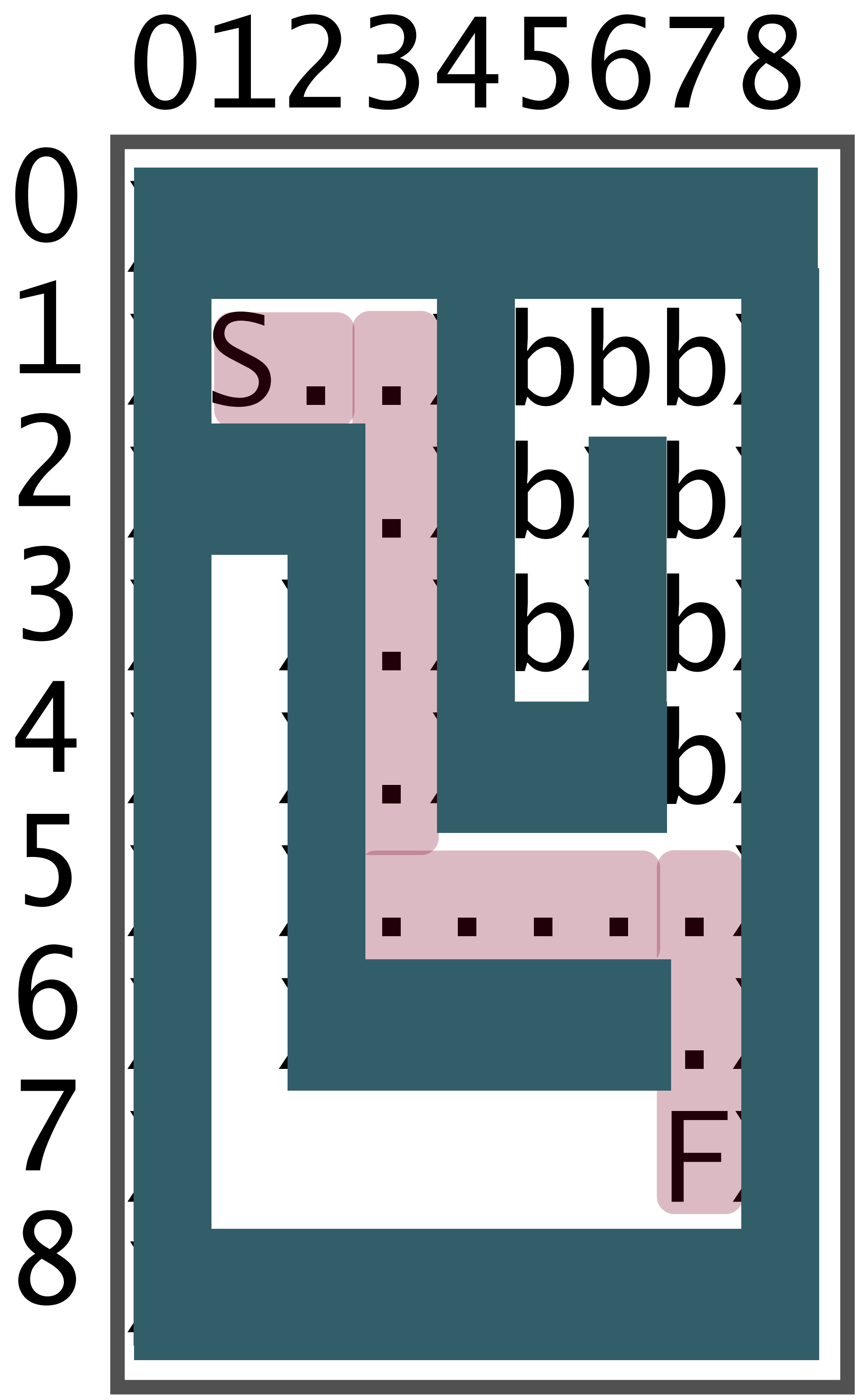
```
bool solveMazeRecursive(int row, int col, Grid<int> &maze) {
        ...
    // north
    if (solveMazeRecursive(row-1,col,maze)) {
        return true;
    }

    // east
    if (solveMazeRecursive(row,col+1,maze)) {
        return true;
    }

    // south
    if (solveMazeRecursive(row+1,col,maze)) {
        return true;
    }

    // west
    if (solveMazeRecursive(row,col-1,maze)) {
        return true;
    }

    maze[row][col] = 'b';
    return false;
}
```



Clumsy Thumbsy: Find All Solutions

You want to write a program that will autocorrect words.

Given a string that represents a single (potentially misspelled) word, a lexicon of English words, a map that maps from a character to a string of the characters near it on a keyboard, and an admissible number of errors, find the Set of all potential intended words.

(Problem courtesy of Jerry Cain)

Clumsy Thumbsy: Find All Solutions

Prototype (note the whitespace -- no need to have this be a giant line!)

```
Set<string> autocorrect(string word,
                        Map<char, string> & nearLetters,
                        Lexicon & dictionary,
                        int maxTypos)
```

First, we have to think of how we will solve this...

Clumsy Thumbsy: Find All Solutions

Prototype (note the whitespace -- no need to have this be a giant line!)

```
Set<string> autocorrect(string word,
                        Map<char, string> & nearLetters,
                        Lexicon & dictionary,
                        int maxTypos)
```

Definition: "maxTypos" : how many letters we can have incorrect

Idea:

- Build up new potential words one character at a time until we have a word (or not).
- Replace all letters with their near-letters.
- Can also choose not to replace a letter!
- Base cases: if we have exhausted our max typos, or if the prefix of the word is not in the dictionary, or if we have built up to a word and it is in the dictionary

Clumsy Thumbsy: Find All Solutions

Prototype (note the whitespace -- no need to have this be a giant line!)

```
Set<string> autocorrect(string word,
                        Map<char, string> & nearLetters,
                        Lexicon& dictionary,
                        int maxTypos)
```

We are going to need a helper function to keep track of the remaining letters, the built-up string, the other reference parameters, and the maxTypos.

```
Set<string> autocorrect(string remaining,
                        Map<char, string> & nearLetters,
                        Lexicon & dictionary,
                        int allowableTypos,
                        string builtUp)
```

Clumsy Thumbsy: Base Cases

```
Set<string> result;
if (allowableTypos < 0 || !dictionary.containsPrefix(builtUp)) {
    // too many typos, or no potential to build word
    return result; //empty set
} else if (remaining == "") {
    if (dictionary.contains(builtUp)) {
        // if word, add it to set
        result.add(builtUp);
    }
    return result;
}
```

Clumsy Thumbsy: Recursive Cases

```
char curr = remaining[0];
string rest = remaining.substr(1);
for (int i = 0; i < (int)nearLetters[curr].length(); i++) {
    result += autocorrect(rest, nearLetters,dictionary,
                      allowableTypos - 1, builtUp + nearLetters[curr][i]);
}

//can also choose not to change character
result += autocorrect(rest, nearLetters, dictionary,
                      allowableTypos, builtUp + curr);
return result;
```

References and Advanced Reading

- **References:**
  - Understanding permutations: http://stackoverflow.com/questions/7537791/understanding-recursion-to-generate-permutations
  - Maze algorithms: https://en.wikipedia.org/wiki/Maze_solving_algorithm

- **Advanced Reading:**
  - Exhaustive recursive backtracking: https://see.stanford.edu/materials/icspacs106b/h19-recbacktrackexamples.pdf
  - Backtracking: https://en.wikipedia.org/wiki/Backtracking

Extra Slides