University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Recursion

Writing the Fibonacci Sequence

Anyone remember the Fibonacci sequence?

1, 1, 2, 3, 5, 8, 13, …

Can anyone write the whole sequence?

(The rest of us can be done for today!)

How about this way: F(0) = 1
F(1) = 1
F(N) = F(N – 2) + F(N – 1)

This answer is a **recursive definition**, a **function defined in terms of itself**.

Fibonacci Sequence is Well-Defined

Fibonacci: F(0) = 1
F(1) = 1
F(N) = F(N – 2) + F(N – 1)

Given this definition, we say that F(N)
- is **well-defined** because
- it **eventually stops recursing** for all N ≥ 0,
- or, **equivalently, F(N)** satisfying the equations **is unique** for all N ≥ 0.

Some Sequences are Not Well-Defined

This sequence is not well-defined:

G(0) = 1
G(N) = [G(N – 1) + G(N + 1)] / 2

What can G(N) be?

1, 1, 1, 1, 1, 1, 1, …

1, 2, 3, 4, 5, 6, 7, …

The possibilities are infinite.

G(N) is not well-defined.

Recursive Functions Must Be Well-Defined for Computers

If you
- write a recursive function
- that is not well-defined,
- **don't expect a computer to choose.**

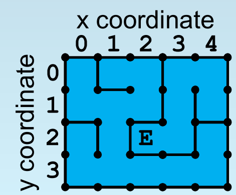
As you know, **computers are dumb**.

Some well-defined recursive functions
- may still be difficult or impossible to express
- in a computer language.

Time for the Today's "Help Prof. Lumetta" Problem!

I need your help again.

If I build a new house
- as a maze …
- … like that …
- **can I go from (0,0)**
- **to the exit at `E`?**



How Did you Solve the Maze?

How did you know?

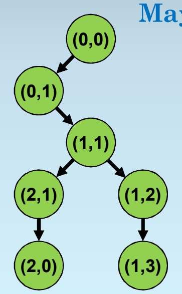
And, more importantly…

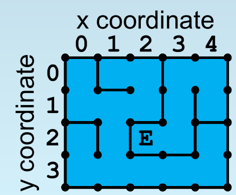
…can you teach my computer?

We Can Use a Tree to Solve the Problem

Maybe you used a tree?

(When we run out of ways to go, we won't have found the exit.)





Represent the Maze with an Array of Bit Vectors

We can **represent the maze with an array**:

```
static uint8_t maze[10][10];
```

Each space in the array **is a bit vector** composed of the following bits:

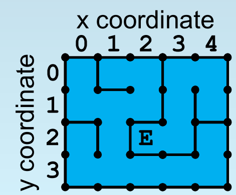
- // **1** – the space has a **left wall**
- // **2** – the space has a **right wall**
- // **4** – the space has an **upper wall**
- // **8** – the space has a **lower wall**
- // **16** – the space is the **exit**

Do You Understand the Representation?

(Reminder: L=1, R=2, U=4, D=8, E=16)

For example,

- `maze[0][0] is 7 (1 | 2 | 4)`
- `maze[0][1] is 9`
- `maze[3][1] is 3`
- `maze[4][2] is 7`
- `maze[2][2] is 29`

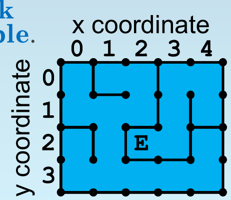


Outline for a Recursive Solution

Let's solve the problem recursively.

Here's the approach:

- **Keep track of reachable locations.**
- Write **a function to mark one location as reachable**.
- Within the function, **call the same function** to mark all "children" (adjacent reachable neighbors) as reachable.



Represent Reachable Locations with a Second Array

Track reachable locations with a second array:

```
static uint8_t found[10][10];
```

Each element is either:

- **0** – the space has **not** been **found**/reached
- **1** – the space has been **found**/reached

And we use one variable for the exit:

```
static int32_t saw_exit;
```

(Both of these should be initialized to all 0s.)

Ready to Write the Recursive Function

Now we're ready to write the function.

Here's a signature:

```
void can_reach (int x, int y);
```

The function should
- **set all locations reachable from (x,y) to 1 in found**, and
- **set saw_exit to 1 iff the exit is reachable from (x,y).**

(To do so, the function will call itself.)

Mark as Reachable, then Check Children

```
void can_reach (int x, int y)
{
    found[x][y] = 1;
    if (0 == (maze[x][y] & 1)) {
        can_reach (x - 1, y);
    }
    if ((0 == maze[x][y] & 2)) {
        can_reach (x + 1, y);
    }
```

(x,y) is reachable.

No left wall?

Space to left is reachable.

Same Check and Marking for Right Child (Value 2)

```
void can_reach (int x, int y)
{
    found[x][y] = 1;
    if (0 == (maze[x][y] & 1)) {
        can_reach (x - 1, y);
    }
    if (0 == (maze[x][y] & 2)) {
        can_reach (x + 1, y);
    }
```

No right wall?

Space to right is reachable.

Same Check and Marking for Upper Child (Value 4)

```
    if (0 == (maze[x][y] & 4)) {
        can_reach (x, y - 1);
    }
    if (0 == (maze[x][y] & 8)) {
        can_reach (x, y + 1);
    }
    if (0 != (maze[x][y] & 16)) {
        saw_exit = 1;
    }
}
```

No upper wall?

Space above is reachable.

Same Check and Marking for Lower Child (Value 8)

```
    if (0 == (maze[x][y] & 4)) {
        can_reach (x, y - 1);
    }
    if (0 == (maze[x][y] & 8)) {
        can_reach (x, y + 1);
    }
    if (0 != (maze[x][y] & 16)) {
        saw_exit = 1;
    }
}
```

No lower wall?

Space below is reachable.

Finally, Check and Mark Exit (Value 16)

```
    if (0 == (maze[x][y] & 4)) {
        can_reach (x, y - 1);
    }
    if (0 == (maze[x][y] & 8)) {
        can_reach (x, y + 1);
    }
    if (0 != (maze[x][y] & 16)) {
        saw_exit = 1;
    }
}
```

Exit here?

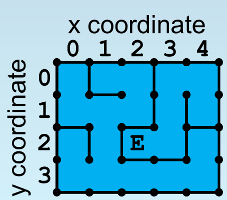
Record having seen exit.

Does Our **can_reach** Function Answer My Question?

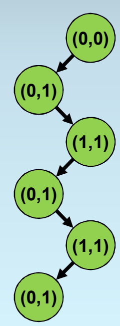
How do we use **can_reach** to answer my question about getting from (0,0) to the exit?

1. Fill **found** and **saw_exit** with 0s.
2. Call **can_reach (0,0)**.
3. Check **saw_exit**.

Does it work?

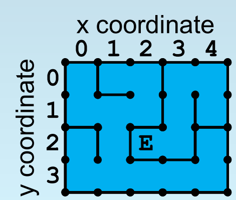


Maybe There's a Bug?



Let's try it!

(Are you still mad about my asking you to write all of Fibonacci?)



When Should We Stop Recursing?

What's the problem?

In math,
- we used base cases
- to stop the recursion.

In a **C** function,
- we **need a stopping condition**
- to stop the recursion.

So: when should we stop?

Stop if the Space Has Already Been Marked as Reachable

Stop if we already reached (x,y).

```
void can_reach (int x, int y)
{
    if (found[x][y]) { return; }
    found[x][y] = 1;
    if (0 == (maze[x][y] & 1)) {
        can_reach (x - 1, y);
    }
```


Review: Induction, Bit-Slicing, Recursion

As mentioned in 120, the following are closely related mathematically
- proof by induction
- bit-sliced hardware design
- recursion.

In all three, one
- solves a small piece of a problem, then
- combines it with the "rest" of the solution
- (which is also solved as small pieces).

A General Strategy for Recursion

Here's a general strategy for recursion.

```
______ recursive ( ______ )
{
    // Check stopping conditions.
    // Handle one node.
    // Handle children.
}
```

These may be swapped.

