Chapter 37

Recursion

This section has only one real rule: a function can call itself. Each time, you get a fresh copy and a fresh set of local variables. That's the whole rule. As you might guess, the real skill is being able to use it to do something useful.

The first part of this chapter is showing the mechanical rules for recursion – no actually useful examples yet. Then next part will be real, but fakey uses of recursion - doing things recursively that we can already do better without it.

Then the last part will be about real problems where recursion is the best way.

Recursion is a tough topic. You have to just "get" it, and even after that it's a tricky thing to do right. But unlike some other hard topics, knowing just a little is useful.

What will happen is: some problem you're trying to solve will naturally break into parts. You'll notice these parts are the same as the original problem, just smaller. And those break into subparts that are the same way, and so on. That's a naturally recursive problem, which means you need recursion to solve it (hopefully by the end of this chapter at least that will make sense.)

Just being able to recognize a recursive problem is a real time-saver, which isn't too hard once you've seen a few examples.

Another funny thing about recursion: you know how with most tricks – like functions, loops and lists – you should have been using them all along? Recursion isn't like that at all. It's pretty much useless except for those few special problems that can't be solved any other way.

37.1 How calling yourself works

A function is allowed to call itself. Just like calling anything else, it waits for the call to end, then continues. The function `crashMe` below is useless, never prints anything and will crash the computer, but it is legal:

```
void crashMe() { crashMe(); print("Arrg"); }
```

When this calls function `crashMe()`, the computer starts running a second fresh copy. The first `crashMe` call is waiting, and the second copy is running. The second copy calls the third copy and waits, and so on.

This makes an infinite number of copies, never getting to line 2 of any of them. A fun fact – each copy takes a little bit of space. So instead of running forever like an infinite loop, it runs out of memory and crashes the entire Unity editor (same as an infinite loop, be sure to save before testing if you changed the Scene.)

Having a chain of `crashMe`'s, all waiting for the next one to finish, isn't the problem. Never stopping is the problem. We can use tricks to make recursion end, just like we use tricks to make loops end.

This next recursive function is still pretty useless, but it will run without crashing:

```
void A(int n) {
  if(n>0) A(n-1); // <- calling ourself
  print(n);
}
```

It works because of the "make a copy of" rule. Each time we call `A`, we get a new copy of it with a new local `n`.

Saying this in a different way: the idea I gave you before about local variables was oversimplified. I made it seem like you could premake one local `n`, which `A` used whenever it was called. The real trick is we really make a fresh local `n` for each call.

Suppose someone calls `A(5)`. It calls `A(4)`, which calls `A(3)` ... down to `A(0)`, which stops because of the `if`. Here's what the call stack looks like after that:

```
 A       A      A       A       A       A
n:5  |  n:4 |  n:3  |  n:2  |  n:1  |  n:0
```

Five copies of `A` are waiting, with their own local `n`'s. The last copy with `n=0` is running. It prints 0 and pops back to `A(1)`. That local `n` is 1. We print that, pop back and print 2 and so on.

A neat thing, pretend you're the first `A(5)`. All you do is call `A(4)`, wait, then print 5. All that growing and shrinking the call stack is handled by the computer. It seems like this is different because the function you're waiting for is you, but it's not.

The whole thing prints 0 1 2 3 4 5, on different lines. To be clear: this isn't a good use of recursion (but it's pretty cool that it works at all.)

If you think back to the old "function calling some other function" examples, it was nice to print some variables just before and just after the call. This standard recursion example does that. We can watch it "go down" and then come back:

```
void B(int n) {
  print("begin: "+n);
  if(n>0) B(n-1); // <- call ourself
  print("DONE: "+n);
}
```

Not even thinking about recursion, we can see `B(3)` will print `begin: 3` first, then some middle stuff, then `DONE: 3` as the last thing. The entire output is:

```
begin: 3 <- first call
begin: 2
begin: 1
begin: 0
DONE: 0 <-finally, we stop calling and start returning
DONE: 1
DONE: 2
DONE: 3 <- leaving first call
```

It's sort of the same logic as when A calls B calls C. The middle part is what `B(2)` does. And the middle of *that* is what `B(1)` does. But even so, yikes!

37.2 Recursive thinking

A recursive function starts with a recursive idea. You have something you can solve by doing a little bit of work to make it the same problem, but smaller. Then you run a copy of yourself solve the smaller version, which repeats the process.

Here's a recursive idea to compute powers of 2: to get 2^n, find the next lower one and double it. For example, 2^4 is two times 2^3. It's recursive since 2^3 is the same problem, but smaller. Another copy of ourself can be used to solve it:

```
int powTwo(int p) {
  if(p<=0) return 1; // 2 to the power of 0 is 1
  else return 2*powTwo(p-1);
}
```

Let's forget about how we can already do this with a loop. Do you see how beautiful that last line is? `return 2*powTwo(p-1)`. It says "call a function to find the next lower power of two, double it, and I'm done."

If `powTwo(3)` computes 8, then `powTwo(4)` will compute 16.

Here's a picture of the largest stack frame if we call `twoPow(4)`. Four functions waiting for the fifth to finish:

```
p:4  |  p:3  |  p:2  |  p:1  |  p:0
```

The fun part is when they start returning things. Each one gets the "one lower" answer, doubles it, and returns that. As the functions pop back, the return values look like the second line:

```
   p:4 | p:3 | p:2 | p:1 | p:0
16 <-  8 <-  4 <-  2 <-  1 <-
```

I think of it as a bunch of guys in a row. Each guy asks the one to the right, then waits. Eventually, they hear the answer, double it, and tell that to the guy who asked them.

You may have realized the "how do I stop" problem is similar to the one we had with loops. So far, we've been counting down to 0, but, like loops, recursion can have all sorts of plans for when to stop.

In this next one I want to pad a string up to a certain length by adding dashes around it. For example `"cow"` padded to 8 would be `"--cow---"`. My recursive plan is this: add a dash on each side and have someone else finish it. And, of course, check if you only need one dash.

The key is how lazy it is. Adding a dash on each end is like a single step of work, and the result is a "more done" version of the same problem. We can hand it off to another version of ourself. The code:

```
string padCenterDashes(string w, int wantSz) {
  print("begin with "+w); // TESTING
  int extraNeeded=wantSz-w.Length;
  if(extraNeeded<=0) return w; // already too big
  if(extraNeeded==1) return w+"-"; // fix and done
  // add one to each side and have my clone finish it:
  return padCenterDashes( "-"+w+"-", wantSz ); // <- recursive call
}
```

Some notes about this:

- A run with `padCenterDashes("cow",8);` calls itself two more times. Once with `"-cow-"`, again with `"--cow--"`. That last one knows to only add a dash to the end.

- Checking this is typical loop-logic. `w` always gets larger (by at least 1), `wantSz` never increases, so this will eventually stop when `w` gets big enough.
- This isn't an example of where we need recursion. It's pretty fake. For real it's easy to have loops add 1/2 the dashes to the front and back. But this way is pretty cool how it avoids math.

You might notice one difference between recursion and loops: when a loop ends, it just ends. Recursion usually ends with an answer. It ends by saying "hey - this input is so easy I can just do it." We call that the **basis step**.

In the power-of-2 functions, the basis step is `p=0`. In this one, there are two basis steps: big enough, and only needs one more.

This next one is another oddball: finding the largest thing in part of a list, using recursion. The plan is to grab the first item, use recursion to find the largest thing in the rest, and compare:

```
int largestInPart(List<int> N, int start, int end) {
  // basis step. if only 1 item, it's the largest:
  if(start==end) return N[start];

  int nFirst=N[start];
  // use my clone to find largest in the rest of the list:
  int nRest=largestInPart(N, start+1, end);
  if(nFirst>nRest) return nFirst;
  else return nRest;
}
```

Like before, this isn't a place where we'd normally use recursion, but it's still neat to see how lazy it is. Each copy only compares 2 numbers.

This shows another funny thing about recursion. This only works since it has `start` as an input. We need it so we can add 1 when we call ourself. This is pretty common. It's also common to add a non-recursive front-end:

```
// front-end. This is not recursive:
int largestInPart(List<int> N) {
  // call the recursive function with the extra inputs:
  return largestInPart(N, 0, N.Count-1);
}
```

For recursive functions we often call that a **driver**. Sometimes it has to set up a lot of variables. And then it often has to decode the answer from the recursive part into what we really wanted. So we thought a fancy name like Driver sounded better.

This last example uses a few tricks and also feels very recursive-y. Suppose we have a sorted list and want to check for a number. Since it's sorted, we can look at the middle number, pick the correct half, and search there. Saying it recursively: to find a number in a list, find which half it has to be in, and search that half.

To cut the list in half, we'll have to use the trick where the recursive part takes `start` and `end`, and a fake front-end driver fills them in:

```
// driver to fill in start/end for the real recursive function:
bool isInList(List<int> A, int num) {
  // may as well check some easy stuff, first:
  if(A.Count==0) return false; // nothing in list
  if(num<A[0]) return false; // everything in list is larger
  if(num>A[A.Count-1]) return false; // everything in list is smaller

  // have to do a real search. run the recursive function:
  return bSearch(A, 0, A.Length-1, num);
}
```

Here's the recursive code, with extra prints:

```
bool bSearch(List<int> A, int start, int end, int num) {
  print("list is "+start+"-"+end);
  // check basis steps: list is size 0 or 1:
  if(start>end || end<0 || start>=A.Count) { // list is empty
    print("size 0 list=not found");
    return false;
  }
  if(start==end) // size 1. Either this is it, or it's not:
    return A[start]==num;

  int mid=(start+end)/2;

  if(num<=A[mid]) { // check first half of list:
    print("searching H1= "+start+"-"+mid);
    return bSearch(A, start, mid, num);
  }
  else { // check second half of list:
    print("searching H2="+(mid+1)+"-"+end);
    return bSearch(A, mid+1, end, num);
  }
}
```

Getting the math correct for the halves is tricky – there are lots of places to have an off-by-one. But those are details. Instead look how beautiful the recursive calls are: `bSearch(A, start, mid, num)` checks the first half and `bSearch(A, mid+1, end, num)` checks the second half.

A new thing is that previously we made exactly the same recursive call every time. Now we choose which one. But the important thing is we're always getting shorter, and closer to being done.

Some notes: 1) This only works if the list is sorted, 2) there's already a built-in BinarySearch in the List class, 3) for real a loop is better: each time would move `start` or `end` and it stops when they touch, 4) it's called Binary Search since it cuts into 2 halves and binary means 2.

There's one more funny thing about this. Suppose we cut it in half 6 times to get down to the smallest list. That means a chain of 6 functions is waiting for us. We know the answer but there's no way to directly send it back.

We return true or false, and everyone else just sends the same answer up the chain. That's what `return bSearch(A, mid+1, end, num)` does – whatever she said, that's my answer to.

It seems wasteful, but there's no other good way.

37.2.1 Tree functions

The computer science term for parents with children, with yet more children, is a tree. A tree is a recursive data structure, which means only recursion is good at navigating one.

Folders with files and more folders inside is a tree. In Unity (or any 3D environment) trees are commonly used to glue objects together.

Suppose we make three long, thin cubes for fingers, positioned against a flat square for a hand. Moving the hand would leave the fingers behind. To glue them to it, drag them in to make them children. You get this picture:

```
hand
  finger1
  finger2
  finger3
```

Now moving or rotating the hand drags the fingers with it. It's a feature. You can move or tilt any of the fingers, and it will follow the hand using that new, adjusted position. For real, animated 3D models have one "skin" and lots of imaginary bones organized in a tree. So you see this sort of thing a lot.

A more realistic tree might start at the body, with 2 hands a head and a tail, which also have several parts:

```
body
    hand1
        finger1
        finger2
    hand2
        finger1
        finger2
        finger3
    head
    tailA
        tailB
            tailC
                tailSpike1
                tailSpike2
```

This is showing how trees are a recursive data structures. `body` is a tree, but `hand1` is also a tree. `head` is a very small tree. The definition of a tree is a parent, with 0 or more children, which are also trees. The definition loops around back to itself.

It sounds like cheating, but it works pretty well: `tailA` has `tailB` inside of it. What are the rules for `tailB`? Well, it is also a tree, so the rules say it has 0 or more child trees in it.

Before using recursion, we need Unity's tree commands. They work off of the `Transform`. `transform.childCount` is an int, which can be 0. `transform.GetChild(n)` gets one child, using the usual 0-based index (three children go from 0-2).

A sample *non-recursive* function to print someone's name and all of their immediate children:

```
void printMeAndKids(Transform tt) {
  print("parent is "+tt.name);
  int childCount=tt.childCount; // this could be 0
  for(int i=0; i<childCount; i++) {
    Transform tChild = tt.GetChild(i);
    print("child "+i+" is "+tChild.name);
  }
}
```

The problem is each child could have more children, or not. Some children could have lots of levels of grand-grand-grand-children.

Here's the super-cool part: each child, by the definition, is another tree. To print everything in a tree, we *call the tree-print function*! The recursive function:

```
void treePrint(Transform tt) {
  print(tt.name):
  int childCount=tt.childCount;
  for(int i=0; i<childCount; i++) {
    Transform tChild = tt.GetChild(i);
    treePrint(tChild); // <-recursive call
  }
}
```

This is so cool. You don't even print the names of your own kids. They're fully trees on their own, so send them to another `treePrint`.

Some notes:

- This is the first problem where recursion is the best solution. Trying this with loops would be a huge mess of nested-nested-nested loops that still wouldn't work on trees deeper than how many loops you used.

- This is the first recursive example that makes multiple recursive calls. That's the main ability that makes recursion useful, and also super complicated.

  A recursive function making one call each time is just a bad way to write a loop (but is still a good way to practice.)

- This still follows normal function calling and stack rules. But, because of the multiple calls, it grows and shrinks. `body` calls `hand1` which calls `finger1`. When finger1 quits we pop back to hand1, as normal. Which calls `finger2`. Then that pops back to hand1, which pops back to body.

  And now body can finally call `hand2`.

  But that's not a new rule. We already knew that function `A` can call `B`, do more stuff, and call `B` again, since it always resumes from where it left off. But yikes does recursion make it seem complicated.

- The basis step (the required thing where it stops calling itself) is 0 children.

- This is the standard way to hit everything in a tree. It's called a Traversal.

- This happens to run down the long way – everything in `hand1` before looking at hand2. Usually we don't care about the order, as long as we hit everything.

  Printing in order of depth (all children, then all grand-children ...) makes as much sense, but there's no easy tweak to do it. Recursive functions can be delicate – they work the way they work

To show those last two points, this will take any tree made of simple objects and turn them all a color. It's a copy of the same code with "change color" where printing was:

```
void setTreeColor(Transform tt, Color cc) {
  // set our color. Bonus check to be sure we're colorable:
  Renderer rr = tt.GetComponent<Renderer>();
  if(rr!=null) rr.material.color=cc;

  int childCount=tt.childCount;
  for(int i=0; i<childCount; i++) {
    Transform tChild = tt.GetChild(i);
    setTreeColor(tChild, cc);
  }
}
```

This next example is to show how recursion can be deceptively hard. Suppose instead my plan was to color each child as I saw it, then make a recursive call only if it had children. That “feels” better and seems like it would run faster, but it has a bug:

```
void colorRecur2(Transform tt, Color cc) {
  int childCount=tt.childCount; // this could be 0
  for(int i=0; i<childCount; i++) {
    Transform tChild = tt.GetChild(i);

    Renderer rr = tChild.GetComponent<Renderer>();
    if(rr!=null) rr.material.color=cc;

    if(tChild.childCount>0) colorRecur2(tChild, cc);
  }
}
```

The bug is it won’t color the main parent (body, in my example.) We could add that, but then it would color everything twice. It’s not an error in this case, but for some things it would be.

A classic tree function is counting everything in the entire tree. It’s dastardly how something this short actually works (I’m also using a shortcut for going through every child):

```
int treeCount(Transform tt) {
  int count=1; // me;
  foreach(Transform tChild in tt) // visit every child shortcut
    count+=treeCount(tChild); // <- recursive call
  return count;
}
```

It doesn’t even add `tt.childCount`! It hits every item, in the usual way, and counts them all as 1.

You can test it recursively! On a childless item it returns 1. If we have four normal children, the loop runs 4 times, adding +1 each time, which makes five. Suppose we have two of those things in one parent. The answer is 1 + 5 (call on 1st child) + 5 (call on 2nd child).

That’s the way to figure out tricky recursive stuff. Test it on a small thing. When you try it on something larger, all of the recursive calls will be on something you just tested.

37.2.2 Connected / Flood Fill

Finding all of the connected spaces in a 2D grid turns out to be another recursive problem. The solution works out to: mark this space and find all of the free spaces touching it. For all of those spaces, repeat.

That's often called a Flood Fill. Various problems can be solved with it: checking whether every space on a map is connected, or checking whether a fence keeps in a cow (do a flood fill from the cow. Check it can't reach the road).

We'll run this in Unity, so we can see the results in color. That means we need a bunch of boring grid code. You can skip straight to the recursive part. The grid details are mostly obvious.

Boring grid set-up: each square is either a wall or open. Open spaces have a Mark, used to indicate reachable spaces. We'll color walls blue, open areas white, and marks green.

Here's a simple class to handle one square. `G` will be our permanent link to the real square, which we'll make later:

```
// holds info about one square in the grid
class GridSq {
  public bool isWall;
  public bool isMarked;
  public GameObject G; // pointer to the real square

  public bool isNewFloor() { return !isWall && !isMarked; }

  public void setMark() { isMarked=true; colorMe(); }

  public void colorMe() {
    Color cc;
      if(isWall) cc=Color.blue;
      else if(isMarked) cc=Color.green;
      else cc=Color.white;
      G.GetComponent<Renderer>().material.color=cc;
  }
}
```

The string-picture trick is a cheap way to describe a grid. This one has some interesting differently-shaped areas. The `o`'s stand for walls:

```
string[] GridPic = {
 "      o   ",
 "   oooo   ",
 "          ",
 " o        ",
 "oo    oo o",
 " o    o o ",
 " ooo  o   ",
 " o o  o   "};
```

Then here's the code to make it all on the screen:

```
public GameObject gridCube; // drag a flat 1x1 Cube prefab here

GridSq[,] Grid; // "square" 2D array (since quick to set up)

void makeGrid() {
  Grid=new GridSq[10,8]; // size of that picture
  Vector3 pos; pos.y=0; // used to place the tiles
  for(int x=0; x<10; x++) {
    for(int y=0; y<8; y++) {
      GridSq gs = new GridSq();
      Grid[x,y]=gs;
      gs.isMarked=false;
      GameObject gg = Instantiate(gridCube);
      gs.G=gg;

      pos.x=-5+1.1f*x; // usual positioning
      pos.z=4-1.1f*y; // putting 0 at top to match how GridPic is written
      gg.transform.position=pos;

      gs.isWall = GridPic[y][x]=='o';
      gs.colorMe();
    }
  }
}
```

Nothing special or new there. As usual, lots of trial and error and off-end errors to get it to line up.

The flood fill is another way-too-short-looking recursive function. Remember the plan is: mark this space; find all of the free spaces touching it and repeat:

```
public int xStart, yStart; // set to the coords of the space to check from

void Start() { colorConnected(xStart, yStart); }

void colorConnected(int x, int y) {
  // quit if off-edge or not a legal space:
  if(x<0 || x>=10 || y<0 || y>=8) return;
  GridSq gs = Grid[x,y];
  if(!gs.isNewFloor()) return;

  gs.setMark();

  // try all four neighbors, who try it some more:
  colorConnected(x-1, y);
  colorConnected(x+1, y);
  colorConnected(x, y-1);
  colorConnected(x, y+1);
}
```

If you run this, and give `xStart` and `yStart` various values, you should see the different areas turn green. A fun thing is maybe you're not sure where (0,0) is. Run it with (1,1) and see which part turns green.

Some notes:

- This only works because of the global grid and the marks. When one copy of the function marks a space, everyone knows we've already been there. Globals are a common recursion trick.

  We didn't have to make marks for trees since there's only one way to get to each thing.

- This code always stupidly walks into illegal spaces. When it hits a dead-end, it makes recursive calls to walk into the 3 walls around it. It even tries to walk back the way it came (it notices that space is marked, and quits right away.)

  It might be nicer if the 4 calls at the bottom checked for being open before making the call. But it's so cool that only checking the space we're in right now also works.

- It follows the normal function call / stack rules. Each square checks left, right, down and up, but not all at once. A square walks left, then that square walks left, and so on. Eventually a square hits a dead-end, pops back, and only then do you walk right.

  The starting square will call "walk left" and might wait a very long time before it gets to walking right.

- The order of left/right/down/up isn't important. It will affect the order things are marked, but the end result will still be everywhere we can reach from the starting space.

- If the start is in an open area, and you could watch it run, it looks terrible. It won't grow a nice circle. Instead it will snake back-and-forth in a long, winding path. But it will eventually fill in every little crack, which is what matters.

37.2.3 Maze walk

Finding a path between one space and another is a variant of flood fill. The difference is you stop if you land on the destination, and need some way to remember the path.

The secret to saving the path is remembering how the call stack works. If we're sitting on a space 6 moves from the start, we're at the end of a chain of 6 functions which each took one step.

The plan to save the path is to start with a global empty list. Whenever we take a step (including the start space) we'll add that (x,y) to the end of the list. Whenever give up on a space, we'll take it off. The saved list of steps will always be exactly the steps in the function chain we're on now.

Here's the code, with set-up:

```
List<Vector2> Path; // (x,y)'s from start

public int xStart, yStart, xEnd, yEnd; // set these

void Start() {
  makeGrid(); // the old grid from before
  Path = new List<Vector2>(); // length 0, so far
  walkMaze();
}

void walkMaze() {
    // call the recursive function:
    bool found = _walkMazeR(xStart, yStart);

    // show the path
    // (this is hacky. setMark is a cheap way to turn green
    if(found) {
      for(int i=0; i<Path.Count; i++) {
        int xx=(int)Path[i].x, yy=(int)Path[i].y;
        Grid[xx,yy].setMark(); // turn it green
      }
    }
    else print("no path");
}

bool _walkMazeR(int x, int y) {
  if(x<0 || x>=10 || y<0 || y>=8) return false;
  GridSq gs=Grid[x,y];
  if(!gs.isNewFloor()) return false;

  // this space is free, test walk here and try to keep going:
  Path.Add( new Vector2(x,y) );
  gs.isMarked=true; // NOTE: doesn't color it

  if(x==xEnd && y==yEnd) return true; // found it

  if(_walkMazeR(x-1, y)) return true; // found the exit. Done.
  if(_walkMazeR(x+1, y)) return true;
  if(_walkMazeR(x, y+1)) return true;
  if(_walkMazeR(x, y-1)) return true;

  // it was all dead-ends, so erase this step from the list:
  Path.RemoveAt(Path.Count-1);
  return false;
}
```

Running this, with hand-entered start and end's, will make a green path between them. But it's still not smart – it still follows the same winding path in an open area. But it works great if you add walls to make it more mazey.

A fun thing to do is watch this work in slow motion. This next thing is the same maze walk, except it always shows the current path. It will always show a path of green squares, matching the current chain of function calls:

```
// new, funner maze:
string[] GridPic = {
  " oo o     ",
  "  o ooo oo",
  " oo o o   ",
  " o  o   o ",
  " o oo ooo ",
  "      o oo",
  " oooo   o ",
  "    o o   "};

public int startX, startY, endX, endY; // hand-enter fun spots

void Start() {
  mazeDone=false; // global used by mazeSearch
  StartCoroutine(mazeSearch2(startX, startY));
}

// helper function:
bool isInMaze(int x, int y) { return x>=0 && x<10 && y>=0 && y<8; }

// global needed to say when to quit:
bool mazeDone;

IEnumerator mazeSearch2(int x, int y) {
  GridSq ms=Maze[x,y];
  ms.setMark();
  yield return new WaitForSeconds(0.2f); // small delay
  if(x==endX && y==endY) {
    mazeDone=true; yield break;
  }

  print(x+","+y); // testing

  // try to walk x-1, x+1, y-1, y+1:
  int x2=x-1, y2=y; // current test space

  if(isInMaze(x2,y2) && Maze[x2,y2].isNewFloor()) {
    yield return StartCoroutine(mazeSearch2(x2,y2));
    if(mazeDone) yield break;
  }

  x2=x+1; y2=y;
  if(isInMaze(x2,y2) && Maze[x2,y2].isNewFloor()) {
    yield return StartCoroutine(mazeSearch2(x2,y2));
    if(mazeDone) yield break;
  }

  x2=x; y2=y-1;
  if(isInMaze(x2,y2) && Maze[x2,y2].isNewFloor()) {
    yield return StartCoroutine(mazeSearch2(x2,y2));
    if(mazeDone)  yield break;
  }

  x2=x; y2=y+1;
  if(isInMaze(x2,y2) && Maze[x2,y2].isNewFloor()) {
    yield return StartCoroutine(mazeSearch2(x2,y2));
    if(mazeDone)  yield break;
  }

  // no path from here. Walk back
  // (whomever who called us will try other directions)
  ms.isMarked=false; ms.colorMe(); // erase color
  yield return new WaitForSeconds(0.2f);
}
```

The Unity trick using an IEnumerator to slow it down is sneaky, and this doesn't save the path anymore. But it's one of the nicer ways to see recursion working. If you run it in a more open area, you'll even see how each call tries to follow the same pattern.

37.3 Errors

It's very, very easy to get an infinite recursion by mistake.

This tree function accidentally make the recursive call on itself, instead of on a child, so spins forever:

```
void doTreeStuff(Transform T) {
  ...

  for(int i=0; i<T.childCount) {
    Transform tChild=T.GetChild(i);
      doTreeStuff(T); // <- opps!! used myself by mistake. Meant to use tChild
  }
}
```

In this one, the driver accidentally calls itself, instead of the recursive function:

```
bool findPath(int xStart, int yStart, int xEnd, int yEnd) {
  // set things up, then call the real function:
  for(int x=0;x<10;x++) // clear marks, reset color, etc...
     ...

     findPath(xStart, yStart, xEnd, yEnd); // <-- OOPS!! meant to call findPathRecursive
     // this spins forever
}

bool findPathRecursive( ... ) { ... }
```