Chapter 35

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 (examples are later.)

That's called recursion. The real trick is being able to make it useful. There are some problems which naturally break into parts and subparts and subsubparts. It works out that a function calling itself is the best way to trace through some of those problems.

The first part of this section is just going over 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.

Most people don't "get" recursion at first. That's fine. If you just mess around with it a little then mostly forget it, you'll see a problem years from now and think "hey - that looks like recursion would work" and can relearn it.

35.1 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);
  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. A loop would be easier and faster. This is all just for demo purposes.

For fun, here's the full useless but standard counting recursion example. It prints 5, 4, 3, 2, 1, 0 as it makes the calls, and then 0, 1, 2, 3, 4, 5 on the way back:

```
void B(int n) {
  print(n);
  if(n>0) A(n-1);
  print(n);
}
```

Even if this wasn't calling itself, we can see `B(5);` prints 5 first, than whatever the function call says to, then prints 5 last.

35.2 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 have a copy of yourself solve the smaller version, which repeats the process.

Here's an 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. Easy, right? How to figure out 2^3? A recursive call to myself. Here's how it looks:

```
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)`:

```
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.

The formal term for when a recursive function stops (by not calling itself) is the **basis step**. Each recursive call should get closer to the basis step, somehow.

So far, I just subtracted 1 and the basis step was when it got to 0, but you can do other things.

For fun, we could rewrite `powTwo` to hand-check for 0-3. Then `n<=3` would be the basis step. This isn't any better – it runs 3 steps faster, but is longer:

```
int powTwo(int p) {
  if(p<=0) return 1;
  else if(p==1) return 2;
  else if(p==2) return 4;
  else if(p==3) return 8;
  else return 2*powTwo(p-1);
}
```

Finding the smallest item in an array is another one we can solve with recursive thinking, even though a loop would be easier. The plan is: compare the last item to the smallest thing in the rest of the list. How to find that? A recursive call. Stop if the list only has one thing.

Here's the code, with extra print statements:

```
int smallestItem(int[] A, int len) {
  if(len<=0) return -999; // empty list has no answer
  else if(len==1) {
    print("returning A[0]="+A[0]);
    return A[0];
  }

  int li=A[len-1];
  print("len "+len+" calling len "+(len-1)+":");
  int smallRest=smallestItem(A, len-1);
  print("len "+len+" comparing "+li+" and "+smallRest);
  if(li<smallRest) return li;
  else return smallRest;
}
```

If we call `smallestItem(A,4)` with `A=[35, 72, 25, 64]` we'd see this output, showing how it makes the calls, then comes back:

```
len 4 calling len 3:
len 3 calling len 2:
len 2 calling len 1:
returning A[0]=35
len 2 comparing 72 and 35
len 3 comparing 25 and 35
len 4 comparing 64 and 25
```

The most fun part is how the length 4 version calls the length 3 one and waits (first line of output.) The last line is when it's done waiting. We've got

the smallest out of the the rest of the array, and can finally compare it to our 64.

The extra `len` input is a common trick. We want an extra variable to fake-shorten the array, so we just add it as an input. Then we make a front-end to fill it in:

```
int smallestItem(int[] A) { return smallestItem(A, A.Length); }
```

Now the user can call `n=smallestItem(A)`, like normal, which calls the real 2-input version. Front-ends that set up the extra inputs for recursive functions are usually called **drivers**.

This next thing is something new that we don't really need recursion for, but it's not too bad a way to solve it. It's about searching for a number in a sorted list. We can solve that the same way we would for real: look at the middle number, figure out which half our number could be in, look at the middle number in that half .... Eventually we'll find it, or narrow where it could be to nothing.

Thinking of this recursively: the smaller problem is to try to find the number in a smaller list. In the previous problem the list was just 1 smaller. Now it will be half as big. I'll use the same trick where I'll use extra numbers to fake-resize the list. Here's the driver:

```
bool isInList(int[] A, int num) { return bSearch(A, 0, A.Length-1, num); }
```

The second and third inputs are the exact first/last indexes of the part we're searching. The driver just fills in 0 and Length-1, which means we start with the entire list.

`num` is what we want to find. The answer is either yes, it's in the list, or no it's not. Here's the code, with extra prints:

```
bool bSearch(int[] A, int i1, int i2, int num) {
  print("list is "+i1+"-"+i2);
  // note: i1==i2 means the list has 1 thing in it. i1>i2 means it's empty:
  if(i1>i2) {
    print("size 0 list=not found");
    return false;
  }

  int mid=(i1+i2)/2;
  if(num==A[mid]) {
    print("found it at "+mid);
    return true;
    }

  if(num<A[mid]) {
    print("searching H1= "+i1+"-"+mid-1);
```

```
    return bSearch(A, i1, mid-1, num);
  }
  else {
    print("searching H2="+mid+1+"-"+i2);
    return bSearch(A, mid+1, i2, num);
  }
}
```

Suppose `A` has size 101 (which is 0 to 100.) The plan is to check `A[50]`. Then, depending if our number is smaller or larger, we'll check either 0-49 or 51-100.

Look how beautiful the recursive calls are: `bSearch(A, i1, mid-1, num)` checks the first half (from the start, to one before the middle,) and `bSearch(A, mid+1, i2, num);` checks the second half (just past the middle, to the end.)

Here's searching for 60 on the list `[12 16 29 36 39 53 57 62 78 80]`:

```
searching H2= 5-9
searching H1= 5-6
searching H2= 6-6
searching H2= 7-6
size 0 list=not found
```

It cuts the list in half three times to get down to 6 through 6. That's where 60 could have been, but it's not. The last step is trying to cut 6 to 6 in half and realizing we're done and can't find it.

Here's the same list, searching for 36:

```
searching H1= 0-3
searching H2= 2-3
searching H2= 3-3
found it at 3
```

This time the `if` walks us through different halves. When we get down to a size-1 list, that's our number. For fun, here's what happens when we look for 62:

```
searching H2= 5-9
found it at 7
```

We cut it in half once, but then luck out when we try again – the middle is exactly our number. It's not a rule that a recursive function has to go all the way every time.

I skipped a lot of testing steps – this was a lot harder to write than I'm making it look. Anything with indexes is super-easy to be off-by-one, or get wrong index math so it spins forever, skips certain positions, or all sorts of wrongness.

But the really cool thing is this follows the recursion idea. On each call, `i1` and `i2` aren't getting smaller, but they're always getting closer together. The list is always getting smaller.

35.2.1 Tree functions

The computer science term for parents with children, with yet more children, is a tree.

The most common one you see is a folder layout. In a 3D environment, we use an even simpler tree to glue items together. If you haven't seen one, here's how it works:

Suppose we make three long, thin cubes for fingers; then another cube for a hand. We can position the fingers against the hand, but we'd like to formally make them part of it. If we drag them into the hand (the same way we'd drag a file into a folder) they become the hand's children:

```
hand
  finger1
  finger2
  finger3
```

The advantage is we can now move the hand and the fingers move with it.

Just to use the built-in tree commands, here's a non-recursive function that prints the parent and kids:

```
void printKids(Transform tt) {
  print("parent is "+tt.name);
  int childCount=tt.childCount; // this could be 0
  for(int i=0; i<childCount; i++) {
    Transform tt2 = tt.GetChild(i);
    print("child "+i+" is "+t2.name;
  }
}
```

`childCount` is just an integer counting your direct children. `GetChild(childNum)` uses 0-based indexes and returns the `Transform` of that child.

Since those two functions run on Transforms, I made the input be one: we could call this with `printKids(gg.transform);`, or use `public Transform t1;` and just call `printKids(t1);`.

The thing is, real trees are usually a lot messier than this. The same one can be shallow, and deep, and have different numbers of kids. Here's a typical messy one: a body with two different fingered hands and a long forked tail:

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

The trick to walking through something like that is that trees have a recursive definition. A tree is one gameObject with 0 or more children, which are trees themselves.

In recursion, we're looking for a natural way to break a problem into identical smaller problems. A tree breaks into 1 parent and a bunch of smaller trees. The basis case is when you have no children. Here's a recursive function doing that:

```
void printAllNames(Transform tt) {
  print(tt.name):
  int childCount=tt.childCount;
  for(int i=0; i<childCount; i++) {
    Transform tt2 = tt.GetChild(i);
    printAllNames(tt2); // <-recursive call
  }
}
```

This has one thing we've never done before: it makes *several* recursive calls each time. That's legal, and is really the main use of recursion. This is the first thing we've seen where recursion is the hands-down best solution.

To get an idea of using multiple calls, think about the first call, to `body`. It prints `body`, then calls `printAllNames(hand1)` and waits. When that's done it makes the call for `hand2`, `head` (that's a short wait) and finally `tailA`. Each time we're saying "you're a tree, print yourself."

The same set-up can turn everything red:

```
void colorRecur(Transform tt, Color cc) {
  Renderer rr = tt.GetComponent<Renderer>();
  if(rr!=null) rr.material.color=cc;

  int childCount=tt.childCount;
```

```
  for(int i=0; i<childCount; i++) {
    Transform tt2 = tt.GetChild(i);
    colorRecur(tt2, cc);
  }
}
```

It's the same recursive set-up, but with color-change where printing was. An interesting thing is that it makes no difference what order we color things. We happen to go through the entire first hand before going to the second. Changing that order is tricky, but we rarely need to.

Just to mess around, suppose we want to color children as soon as we see them in the loop; then only jump down to them if they have kids. We can do that. We need a driver to color the original parent, so I renamed the recursive part with an underscore:

```
// driver:
void colorRecur2(Transform tt, Color cc) {
    Renderer rr = tt.GetComponent<Renderer>();
    if(rr!=null) rr.material.color=cc;
    _colorRecur2(tt, cc);
}

// recursive part:
void _colorRecur2(Transform tt, Color cc) {
  int childCount=tt.childCount; // this could be 0
  for(int i=0; i<childCount; i++) {
    Transform tt2 = tt.GetChild(i);

    Renderer rr = tt2.GetComponent<Renderer>();
    if(rr!=null) rr.material.color=cc;

    if(tt2.childCount>0) _colorRecur2(tt2, cc);
  }
}
```

This is a bunch more complicated. I'm writing it out only to show what a puzzle it can be to write recursive functions. There's often a short way that looks totally obvious; but it can take hours of trying head-hurting messy junk to come up with it.

Unity has a super-hacky shortcut for finding children. A `foreach` loop takes a list as input, hitting every item. But C# has a trick for it – if you're not a list, you can fake-up a list that `foreach` will use. Unity did that for Transforms – they're specially written to give `foreach` a list of their children.

It's a total hack, and seriously weird-looking, but `foreach(Transform tt2 in tt)` will make `tt2` hit every one of `tt`'s children.

Here's something using that shortcut to give every finger a random length:

```
void resizeFingers(Transform tt) {
  string w=tt.name;
  if(w.Length>=6 && w.Substring(0,6)=="finger") {
    Vector3 sc=tt.localScale;
    sc.z=Random.Range(0.6f, 1.4f);
    tt.localScale=sc;
  }

  // new super-short "call each child":
  foreach(Transform tt2 in tt) {
    resizeFingers(tt2);
  }
}
```

Here's one final classic tree function, which just counts everything in the entire tree. It's dastardly how something this short actually works:

```
int treeCount(Transform tt) {
  int count=1; // me;
  foreach(Transform tt2 in tt)
    count+=treeCount(tt2);
  return count;
}
```

35.2.2 Connected / Flood Fill

This section is about a 2D grid, like a dungeon map or a maze, where we fill in squares to make walls. We often want to know whether we can walk between all of them – whether we can or can't. For example, maybe we're making a random map and want to be sure it's all connected. Or the game is to make a fence around a cow and we're checking to be sure it can't walk out.

Either way, the basic trick is to pick 1 square and then find every other square we can reach from it. The generic name for that is a flood-fill.

I'll start with the boring set-up for the grid. Each square is either a wall, or open. For now, we'll just color walls blue and open areas white. 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 canWalk() { 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 variable `isMarked` is a common trick. It doesn't mean anything for real – it's an extra scratch variable to "mark" a space, which we'll need later.

I'll use a string-picture as a cheap way to describe the grid. This one has some interesting differently-shaped areas. The `o`'s are walls:

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

GridSq[,] Grid;

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

Then here's the code to make the grid 2d-array, and build it on the screen. I'm assuming `gridCube` is a flatish 1x1 cube:

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

GridSq[,] Grid;

void makeGrid() {
  Grid=new GridSq[10,8];
  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.

An interesting thing about finding every place we can go, is that it doesn't seem like a recursive problem, and it really isn't. It feels like you'd use maybe a nested loop or something. It just turns out, after hours and days of trying, a recursive trick is the best way to solve it.

The recursive plan is: mark the square you're on, then make recursive calls to the squares all around you. But just quit if you're already marked. As usual, it's crazy that it works, and how short it is for what it does.

`colorConnected` is the recursive function:

```
public int xStart, yStart;
void Start() { colorConnected(xStart, yStart); }

void colorConnected(int x, int y) {
  // just 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.canWalk()) return;

  gs.setMark();

  // all four neighbors:
  colorConnected(x-1, y);
  colorConnected(x+1, y);
  colorConnected(x, y-1);
  colorConnected(x, y+1);
}
```

Noticed I used the trick where you do the work at the top of the code, for just that one spot. Down below, the code can jump inside walls or off the edge

without checking.

This should mark (and turn green) all the squares you can get to from position `(xStart,yStart)`.

35.2.3 Maze walk

The flood-fill trick can be used to find a path through a maze. The basic idea is the same – take one step in each direction and try from there. The difference is we want to stop when we reach the end.

To do that, each call in a direction returns true if it found the exit. If it does, we don't try any other directions.

Here's the code, then some notes. The `mazeWalk` part is just making sure the marks are cleared, calling the real function, then printing the results. `_walkMazeR` is the real recursive function. You should be able to see it looks about like the flood-fill one:

```
int[] xWalk, yWalk;
int walkLen=0; // this is also where the next grid spot goes

void Start() {
  makeGrid(); // from the old grid
  xWalk = new int[8*10]; // enough for every space in the maze
  yWalk = new int[8*10];
  walkMaze();
}

void walkMaze() {
  // clear everything for fresh walk:
  walkLen=0;
  for(int i=0; i<10; i++)
    for(int j=0; j<8; j++) {
      Grid[i,j].isMarked=false;
      Grid[i,j].colorMe();
    }

    // call the function:
    bool found = _walkMazeR(xStart, yStart);

    // Use the list however we like. Just print it, for now:
    if(found) {
      for(int i=0; i<walkLen; i++) {
        int xx=xWalk[i], yy=yWalk[i];
        Grid[xx,yy].setMark(); // cheap way to turn it blue
      }
```

```
    }
    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.canWalk()) return false;

  // this space is free, test walk here and try to keep going:
  xWalk[walkLen]=x; yWalk[walkLen]=y; walkLen++;
  gs.isMarked=true;

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

  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:
  walkLen--;
  //gs.isMarked=false; // no need, and may as well warn others this is a dead-end
  return false;
}
```

Notes:

- I'm using global int-arrays `xWalk` and `yWalk` to save the actual path. Each time we make a call, we add that (x,y) to the path. Each time we give up (all 4 directions were no good,) we take ourself off the list. It's a sneaky, but common trick.

  After the call is made, at the end of non-recursive driver `walkMaze`, a loop just uses them to color the path.

- We still need the marks, to keep us from walking in circles. But they aren't the answer anymore. For example, I leave the marks on dead-ends as an "I already tried this" if we come to it some other way. I sort of cheated using `gs.isMarked=true;` to avoid changing the color.

- As soon as we find the exit, we can quit the whole thing. But there's no good way to say "pop back to before all the recursion." Instead, we have to pop back with `true` one step at a time, like this does.

- This still works like a flood-fill, wandering around however. That works fine for a "tight" maze. But if you have open areas, like a 3x4 wall-less part, this will stupidly wind around in a lot of extra steps.

You can change this to try every possibility and remember the shortest, but that's more work.

About that last point, the easiest way to see how we're really stupidly wandering in a pattern is the watch a maze-walk run slowly.

If you've seen the way Unity times things using `IEnumerator` (I haven't covered it, at all,) here's a version which walks a maze at 5 steps/second, coloring as it goes. It's the same code, but hacked to get delays in it:

```
string[] MazePic = {
  " 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;

// driver. Calls recursive function with starting coords. Not 100% needed:
void mazeSearch() {
  mazeDone=false;
  StartCoroutine(mazeSearch2(startX, startY));
}

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

// global
bool mazeDone=false;

IEnumerator mazeSearch2(int x, int y) {
MazeSq ms=Maze[x,y];
ms.setMark();
yield return new WaitForSeconds(0.2f);

if(x==endX && y==endY) {
mazeDone=true; yield break;
}

Debug.Log(x+","+y);

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

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

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

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

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

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

35.3 Errors

Obviously, lots of possible errors. My favorites are infinite loops caused by calling the wrong function or using the wrong value.

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 t2=T.GetChild(i);
      doTreeStuff(T); // <- opps!! used myself by mistake. Meant to use t2
  }
}
```

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

```
bool findPath(int xStart, int yStart, int xEnd, int yEnd) {
  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( ... ) { ... }
```