Chapter 10: Recursion

10.0.1 Objectives

The chapter provides a brief introduction to recursion, which is the practice of using a function that calls itself. Students will learn what recursion is and how to read recursive code. Students will learn the key components of a recursive program and, importantly, they will learn that any recursive program must have a means of exiting the function via a base case.

10.0.2 Topic Outline

10.0.3 Key Terms

Recursive functions	Base case
Terminating condition	Infinite recursion

10.0.4 Key Concepts

A **recursive** function is a function that calls itself. Recursive functions are useful for working with **self-similar problems**. There are two ways of thinking of recursion:

- Recursion **reduces** a problem to smaller, similar, problems that can be solved more easily than the larger problem.
- Recursion **expands** computation by repeating a smaller, similar procedure as part of carrying out a larger procedure.

A recursive function has a conditional for dealing with two different cases:

- The **recursive case**. In this case, the function calls itself to solve a smaller problem then use that solution to solve the complete problem.
- The **base case**. In this case, function recognizes the simplest situations and completes the computation without calling itself.

Both cases are important. Since working out the recursive case usually takes a lot of thinking, programmers often forget about the simple base case. When that happens, a recursive function will call itself repeatedly in an infinite recursion and the function will never complete (until a debugger interrupts it).

The best way to understand recursion is with examples.

A Triangular Spiral Example

Previously we have created computer algorithms by thinking about the first steps first, but in recursion, the planning is done from the end first. Imagine that most of the problem has already been solved. With recursion, the programmer asks: once almost everything is done, how would the very last step be solved? A recursive algorithm is built by starting by just thinking about just this last step.

Suppose a recursive program needs to measure a triangular spiral where each side is 10 units longer than the previous side, with the first side 10, then the next side 20, then the next side 30, and so on. How long is a spiral with N sides?

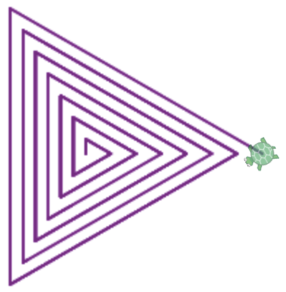
Solving this problem using recursion requires two cases.

- The **recursive case**. To make a spiral with N sides, imagine that the program can already solve the problem for size N - 1. Call the answer spiral(N - 1). Since the last side has length 10 * n, we therefore know that spiral(N) = spiral(N - 1) + (10 * N).
- The **base case**. Some N is needed which does not require on self-reference. It is convenient to do this when N is zero, as spiral(0) = 0.

To write this in code, use an “if” conditional.

```
function spiral(N) {
  if (N > 0) {                 // The recursive case.
    var len = spiral(N - 1); // Assume we can do a smaller spiral.
    fd(10 * N);                // Now draw just the last leg of the big spiral.
    rt(120);
    return len + (10 * N);     // Return the whole length.
  } else {
    return 0;                  // The base case is zero: zero length.
  }
}
pen(purple);
spiral(20);
```

Output

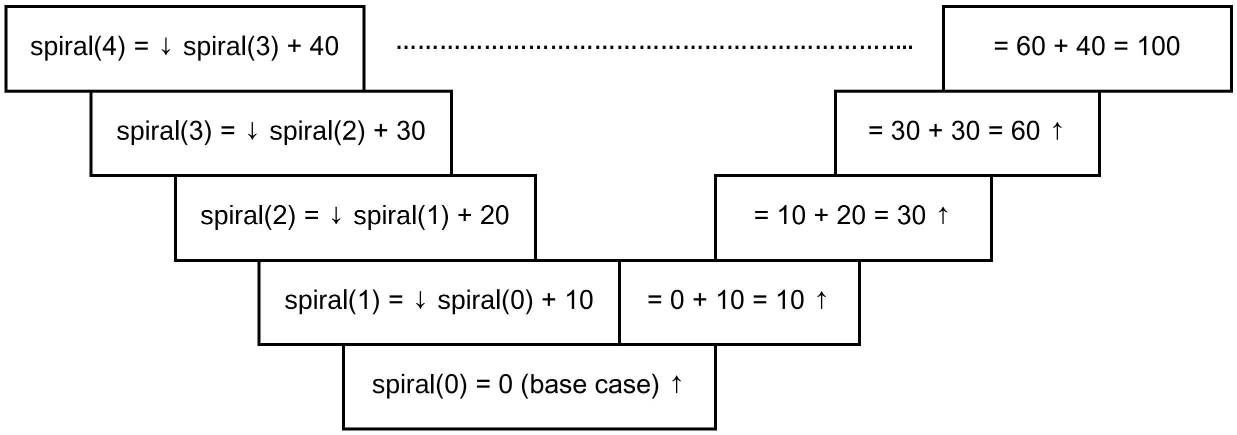


A recursive program to measure a triangular spiral.
The spiral function calls itself and then adds just the last leg.

In thiscode, `spiral(20)` makes a call to `spiral(19)`, makes a call to `spiral(18)`, and so on. How can this possibly work? The computer does not actually have the answer to `spiral(19)` before running `spiral(20)`!

Tracing Out Recursion on a Grid

Every time a function is called, its parameters can take on a different meaning, and so a computer can even call `spiral(3)` even if the invocation of `spiral(4)` is still not completed. The variable N means something different in each invocation and the “layers” do not interfere with each other. The computation layers in the diagram below show how this works.


spiral(4) = ↓ spiral(3) + 40
= 60 + 40 = 100
spiral(3) = ↓ spiral(2) + 30
= 30 + 30 = 60 ↑
spiral(2) = ↓ spiral(1) + 20
= 10 + 20 = 30 ↑
spiral(1) = ↓ spiral(0) + 10
= 0 + 10 = 10 ↑
spiral(0) = 0 (base case) ↑

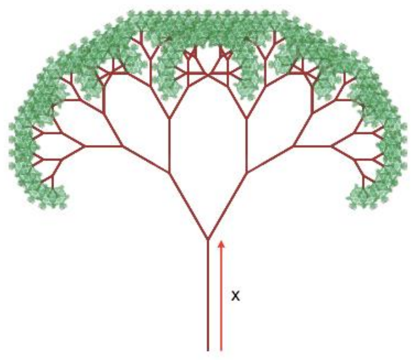
Each row is a single level of the recursion, showing how it calls the next level of the recursion, with the base case at the bottom. Moving from top to bottom shows how each layer of recursion reduces the problem to a smaller problem.

Reading from left to right reveals how the computation proceeds over time: `spiral(4)` does a function call to `spiral(3)`, which does a function call to `spiral(2)`, and so on until the base case returns 0, which allows `spiral(1)` to complete and return 10, which allows `spiral(2)` to complete and return 30, which allows `spiral(3)` to complete and return 60, which allows `spiral(4)` to complete and return 100.

A Fractal Tree Example

Recursion can be used to create self-similar patterns. If a recursive function calls itself twice, then a branching effect can be created, where a single function call expands to 2, 4, 8, etc, calls. In the example below, the recursive function `tree` takes two inputs, a turtle object `t` and a branch length `x`.

```
function tree(t, x) {
  t.fd(x);          // 1. Draw a line
  if (x < 10) {
    return;         // 2. Base case
  }
  var r = t.copy(); // 3. Copy the turtle
  t.lt(30);
  tree(t, x * 0.7); // 4. Left mini-tree
  r.rt(30);
  tree(r, x * 0.7); // 5. Right mini-tree
}
pen(brown);
tree(turtle, 100);
```



An explanation of the portions of this program.

1. The only drawing done directly is to draw one line of length x by moving forward `t.fd(x)`.
2. Then the base case is handled: if the line was shorter than 10, it returns with no further action.
3. A branch will be made by making a copy of `t` called `r`. This will handle the right side.
4. The original turtle t will handle the left side by turning left 30 degrees and drawing another tree. The tree on the left will begin with a branch that is smaller than x by multiplying by 0.7.

5. The new turtle r will handle the right side by turning right 30 degrees and drawing another tree.

The tree in the example above is self-similar because a large tree is made up of smaller trees. Many shapes in nature have the same kind of self-similarity, and recursive programs like this can be used to create organic, natural shapes.

10.1.1 Teaching Suggestions

Students often find recursion intimidating, so teachers often need to revisit this concept several times to make sure the students truly understand it. In an introductory programming class, students should at least be exposed to the idea of recursive functions and understand how recursive functions work. It helps a great deal if they can see recursion in action. It can be helpful to ask students to solve a couple of recursive problems using paper and pencil to predict the output. At this point we do not recommend that students create their own recursive solution. You can use the two Pencil Code examples provided in this lesson as a tool explain and demonstrate recursion. It is best to use multiple strategies to teach recursion.

10.1.2 Possible Timeline: 1 55-minute class period

Instructional Day	Topic
1 Day	Lesson Plan I
2 Day	Lesson Plan II

10.1.3 Standards

CSTA Standards	CSTA Strand	CSTA Learning Objectives Covered
Level 3 A (Grades 9 – 12)	Computational Thinking (CT)	Explain how sequence, selection, iteration, and recursion are building blocks of algorithms.

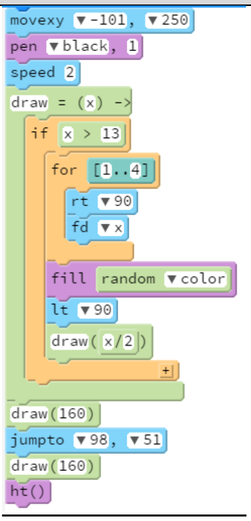
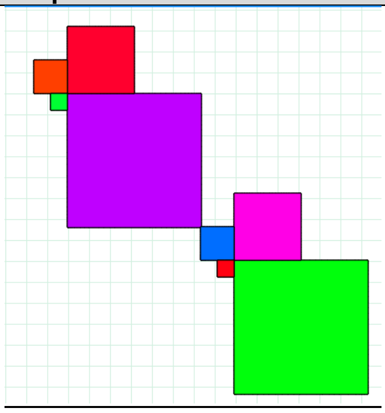
10.1.4 Lesson Plan I

This lesson shows the recursion process. The demonstration should take about 30 minutes. After the demonstration, give the students a copy of the program and let them play experiment with to expand their understanding of how the recursive functions work.
Teaching Notes:
Invariably this is the hardest topic in programming for beginner programmers. The visual feedback that Pencil Code provides helps with understanding the concepts better. Below is a suggested two-fold approach.

First step: demonstrate the program
Next: break it down into pieces and explain the concept using smaller snippets of code.

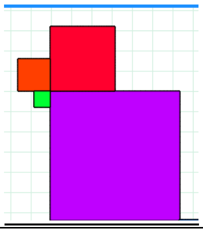
Demonstrate the program on the projector (5 minutes)

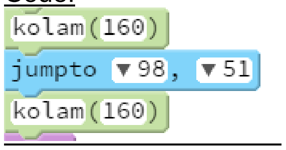
Code	Block	Output

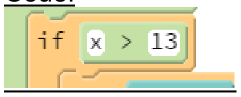
`movexy -101, 250`
`pen black, 1`
`speed 2`
`kolam = (x) ->`
`  if x > 13`
`    for [1..4]`
`      rt 90`
`      fd x`
`    fill random color`
`    lt 90`
`    kolam(x/2)`
`kolam(160)`
`jumpto 98, 51`
`kolam(160)`
`ht()`
	
	`movexy -101, 250`
	`pen black, 1`
	`speed 2`
	`draw = (x) ->`
	`  if x > 13`
	`    for [1..4]`
	`      rt 90`
	`      fd x`
	`    fill random color`
	`    lt 90`
	`    draw(x/2)`
	`draw(160)`
	`jumpto 98, 51`
	`draw(160)`
	`ht()`
		

Demonstrate the concept taking small snippets of code

Content details	Teaching Suggestions	Time

Code
`kolam = (x) ->`
`  if x > 13`
`    for [1..4]`
`      rt 90`
`      fd x`
`    fill random color`
`    lt 90`
`    kolam(x/2)`
	Step 1:
	The rectangles keep getting smaller at each call to the function.
	Teaching Tip:
	Project this code and emphasize that the value of x keeps getting smaller.
	
		Demonstration:
		15 minutes

Code:

`kolam(160)`
`jumpto 98, 51`
`kolam(160)`
 	
`kolam(160)`
`jumpto 98, 51`
`kolam(160)`
	Step 2:
	The main program calls the function twice and generates two sets of three squares.	

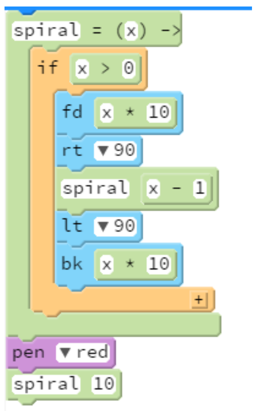
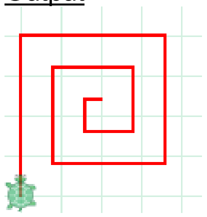
Code:

`if x > 13`
 	
if x > 13
	Step 3:
	There is a condition that is checked before the function is called again.	

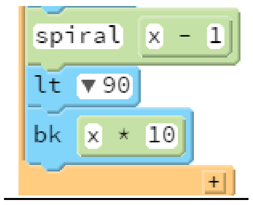
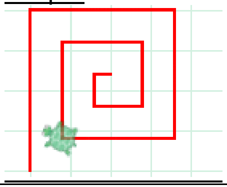
	Step 4: When the condition fails, the program exits out of the recursive cycle and the control is returned to the main program	

10.1.5 Lesson Plan II

This lesson plans demonstrates the recursive process and shows how the complier stacks the commands that are not yet executed. This should take about 20 minutes to demonstrate and explain.

Content details	Teaching Suggestions	Time

, Code:
`spiral = (x) ->`
`  if x > 0`
`    fd x * 10`
`    rt 90`
`    spiral x - 1`
`    lt 90`
`    bk x * 10`
`pen red`
`spiral 10`

`spiral = (x) ->` `if x > 0` `fd x * 10` `rt ▼90` `spiral x - 1` `lt ▼90` `bk x * 10` `+` `pen ▼red` `spiral 10`
	Demonstrate the program and explain the recursive function.
	
	Point out the recursive formula and the condition that helps end the recursion.
	
	Emphasize the importance of the conditional statement.
	
	Output
	
		Demonstration:
		25 minutes

Code Snippet:
`spiral x - 1`
`    lt 90`
`    bk x * 10`

`spiral x - 1` `lt ▼90` `bk x * 10` `+`
	When the recursive function is called, the two statements after the recursive call are stacked.
	Once the recursive calls ends the two statements are popped out of the stack.
	This is illustrated by the turtle tracing the spiral in reverse order.
	Output
		

Encourage students to play execute various recursive programs that are available in the resources list. Students should be encouraged to change values and see the effect of the changes.		Student Practice: 55 minutes