1. The `turtle` library in Python is available for drawing pictures using *turtle graphics*. Imagine a robotic "turtle" starting in the center of the *x*-*y* plane at position (0, 0), facing east. The turtle has a "pen" in its hand, which it uses to draw lines as it moves around the plane. Start Python and type the following two commands:

```
from turtle import *
reset()
```

   This creates the graphics window and centers the turtle in the window. Typing `forward(100)` moves the turtle 100 pixels forward, while `backward(50)` moves it backward 50 pixels. To rotate the turtle left 30 degrees, type `left(30)`. You can also abbreviate these commands as `fd(100)`, `bk(50)`, and `lt(30)`. Here is a list of the main turtle commands and their abbreviations:

Command	**Abbreviation**	**Action performed**
`forward(`*distance*`)`	`fd`	move forward
`backward(`*distance*`)`	`bk`	move backward
`left(`*degrees*`)`	`lt`	left turn (counterclockwise)
`right(`*degrees*`)`	`rt`	right turn (clockwise)
`setheading(`*degrees*`)`	`seth`	set turtle's heading to the given angle
`goto(`*x*, *y*`)`	-	move to position (*x, y*)
`home()`	-	move to position (0, 0) and set heading to 0 (east)
`circle(`*radius*`)`	-	draw a circle starting from the current position/heading
`dot(`*size*, "*color name*"`)`	-	draw a circular dot of diameter *size* in the given color
`penup()`	`up`	lift the pen up (no drawing will occur when moving)
`pendown()`	`down`	put the pen down (drawing will occur when moving)
`pencolor("`*color name*`")`	`color`	set the color of the pen
`pensize(`*width*`)`	`width`	set the thickness of the pen in pixels
`reset()`	-	reset the turtle and drawing window to the initial state
`showturtle()`	`st`	make the turtle visible
`hideturtle()`	`ht`	make the turtle invisible

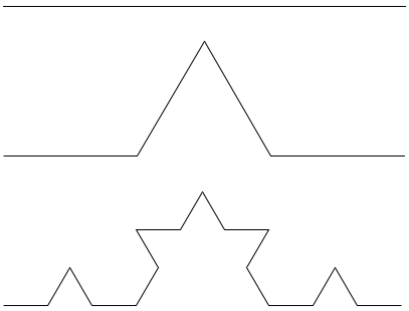
   Experiment with these commands interactively, until you are comfortable with them.

2. Download the file **lab14.py** from the class web page (under Labs), open it in IDLE, and examine the `spiral` function definition. Then test it out by calling `spiral(10, 20)` and `spiral(50, 5)`.

3. You'll notice that it takes some time for the turtle to finish the drawing. We can speed up the turtle with the command `speed(`*value*`)`, where *value* is a number from 0 to 10. Speed 0 is the fastest, speed 1 is the slowest, and values from 2 to 10 gradually increase in speed. Add the line `speed(5)` immediately after the `reset()` command and run the program again. (Calling `reset()` automatically resets the speed to 3, so make sure to put the call to `speed` after `reset`.) Try other speeds for comparison, including 0, 1, and 10. You can find out more info on the speed command by typing `help(speed)`.

4. As another example, the function `drawface(`*x, y, diameter*`)` will draw a simple "face" of a specified diameter, centered at location (*x, y*) in the window. This program illustrates the use of the `dot` command to draw circles of different sizes, and the `pensize` command to change the thickness of the line drawn by the turtle. Notice also that after drawing the face, the turtle moves back to location (*x, y*) and leaves the pen in the "up" state. Test out this function by calling `drawface(0,0,300)` and `drawface(-200,-200,150)`.

5. We can use turtle graphics and recursion to draw some interesting geometric curves. One famous example is the Koch curve (pronounced "coke"). The Koch curve is described in terms of "levels". A level-0 curve is just a straight line segment. A level-1 curve is formed by placing a "bump" in the middle of the line segment, and a level-2 curve is formed by placing bumps on each of the level-1 segments, as shown on the next page:

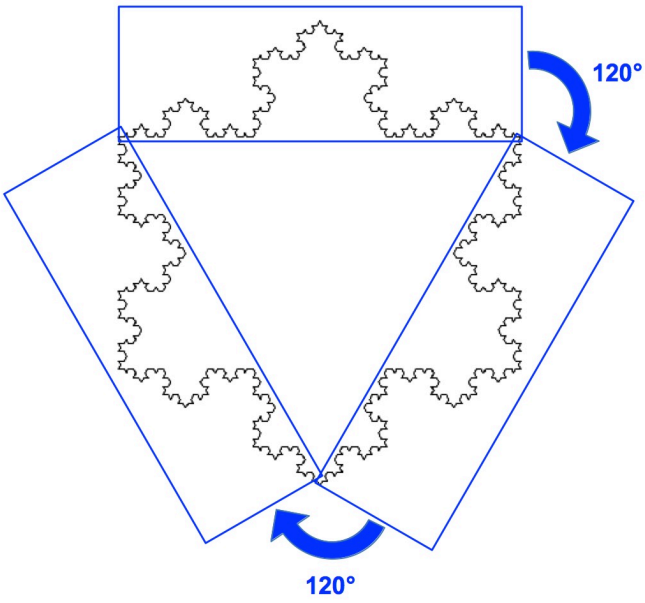


Continuing this process forever produces the full Koch curve. Notice that the original straight line segment gets replaced by four smaller pieces, each of which is 1/3 the length of the original. The bump rises at 60 degrees, and forms two sides of an equilateral triangle. Write a recursive function called **drawkoch(*length*, *n*)** that draws a level-*n* Koch curve *length* pixels wide. For example, `drawkoch(300, 2)` should produce the third curve shown above. Here is an outline of the recursive algorithm:

To draw a level 0 Koch curve (the base case), the turtle should just draw a single straight line of size *length* by moving forward *length* pixels.

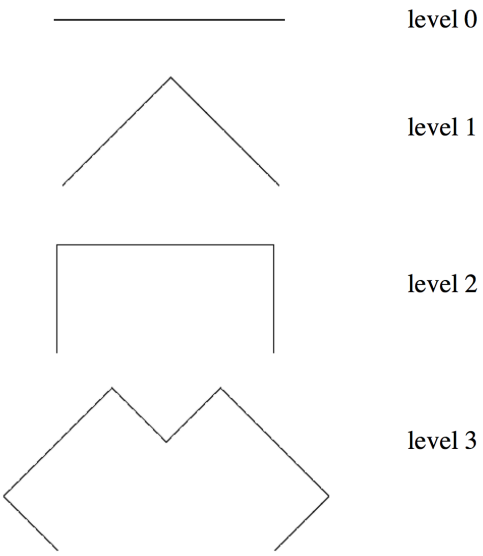
To draw a level *n* Koch curve (the general case), the turtle should:
- recursively draw a level *n*–1 Koch curve of size *length*/3
- turn left 60 degrees
- recursively draw a level *n*–1 Koch curve of size *length*/3
- turn right 120 degrees
- recursively draw a level *n*–1 Koch curve of size *length*/3
- turn left 60 degrees
- recursively draw a level *n*–1 Koch curve of size *length*/3

6. If you draw three level-*n* Koch curves at 120-degree angles, you get a level-*n* "Koch snowflake":

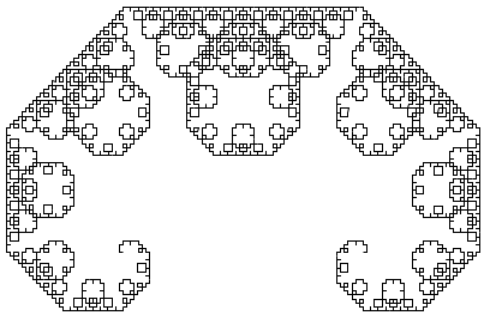


Using `drawkoch` as a helper function, define a new (non-recursive) function called **snowflake(*n*)** that first calls `reset()` and then draws a level-*n* Koch snowflake. For example, calling `snowflake(4)` should produce the level-4 snowflake shown above. To greatly speed up the drawing process, try adding the command `tracer(0)` to the very beginning of your snowflake function to turn off the little turtle icon before drawing anything, and the command `tracer(1)` to the very end of the function, to turn the icon back on after drawing is complete.

7. Another interesting recursive curve is the C-curve. It is formed in a similar way to the Koch curve except that instead of replacing each segment by four smaller segments of *length*/3, the C-curve replaces each segment by two segments of *length*/sqrt(2) that form a 90-degree "elbow". The first few levels are shown below:



Here is a level-12 C-curve generated from an initial line segment of length 200:



Write a recursive function called **drawC(*length*, *n*)** that draws a level-*n* C-curve of size *length*. Your `drawC` function should do the following:

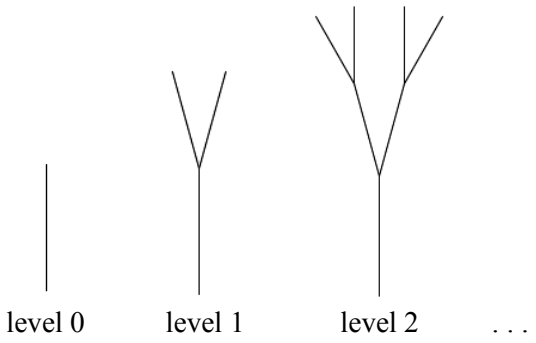
To draw a C-curve of level 0, just draw a straight line segment *length* pixels long.

To draw a C-curve of level *n*:
- turn left 45 degrees
- recursively draw a level *n*−1 C-curve of size *length*/sqrt(2)
- turn right 90 degrees
- recursively draw a level *n*−1 C-curve of size *length*/sqrt(2)
- turn left 45 degrees

Here is a test program for testing your `drawC` function with different values of *n*. Try it out with *n* = 13.

```
def hair(n):
    tracer(0)
    reset()
    drawface(0, 0, 300)
    goto(-100, 0)
    down()
    drawC(200, n)
    tracer(1)
```

8. We can also draw recursive "trees", as follows. A level-0 tree consists of a single "trunk", with no branches. A level-1 tree has a trunk, with two symmetric branches at the top, each of which are level-0 trees consisting of slightly shorter line segments. The branches of a level-2 tree are slightly shorter level-1 trees, and so on:



Write a recursive function called **drawbranch(*length*, *n*)** that draws a tree as follows:

To draw a level 0 tree:
move the turtle forward to draw a line segment of size *length*
move the turtle backward one *length* so that it returns to its starting point

To draw a level *n* tree:
draw a line segment of size *length* (the trunk)
turn left 15 degrees
recursively draw a level *n*–1 tree of size *length**0.8 (the smaller left branch)
turn right 30 degrees
recursively draw a level *n*–1 tree of size *length**0.8 (the smaller right branch)
turn left 15 degrees
move the turtle backward one *length* so that it returns to its starting point

You should also define the function **tree(*n*)** shown below, which clears the drawing window and repositions the turtle at (0, –300), near the bottom of the window, with a heading of 90 degrees (north), and then calls `drawbranch` to draw a level-*n* tree with an initial branch length of 100 pixels.

```
def tree(n):
    tracer(0)
    reset()         # clear drawing window
    up()            # pen up
    goto(0, -300)   # reposition turtle at bottom of window
    setheading(90) # point the turtle due north
    down()          # pen down
    drawbranch(100, n)
    tracer(1)
```

9. Once your tree-drawing program works, experiment with different values for the trunk length, branching angle, and "shrinkage" parameters (the factor 0.8 in the algorithm above). You could also try drawing more than two branches. Also, try drawing a small green dot (a "leaf") at the end of each level-0 line segment.

   Another idea that can make your trees look much more realistic is to introduce a bit of random variation by choosing the shrinkage factor randomly from a range of values, such as from 0.7 to 0.9, instead of always using 0.8. You could also try adding some random variation to the turn angles, or changing the thickness of the pen depending on the level of recursion. How lifelike can you make your recursive trees look?